A Cost Model for a Graph-Based Intermediate-Representation in a Dynamic Compiler[*]

David Leopoldseder
Johannes Kepler University Linz
Austria
david.leopoldseder@jku.at

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

Manuel Rigger
Johannes Kepler University Linz
Austria
manuel.rigger@jku.at

Thomas Würthinger
Oracle Labs
Zurich, Switzerland
thomas.wuerthinger@oracle.com

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

Abstract

Compilers provide many architecture-agnostic, high-level optimizations trading off peak performance for code size. High-level optimizations typically cannot precisely reason about their impact, as they are applied before the final shape of the generated machine code can be determined. However, they still need a way to estimate their transformation's impact on the performance of a compilation unit. Therefore, compilers typically resort to modelling these estimations as trade-off functions that heuristically guide optimization decisions. Compilers such as Graal implement many such handcrafted heuristic trade-off functions, which are tuned for one particular high-level optimization. Heuristic trade-off functions base their reasoning on limited knowledge of the compilation unit, often causing transformations that heavily increase code size or even decrease performance. To address this problem, we propose a cost model for Graal's high-level intermediate representation that models relative operation latencies and operation sizes in order to be used in trade-off functions of compiler optimizations. We implemented the cost model in Graal and used it in two code-duplication-based optimizations. This allowed us to perform a more fine-grained code size trade-off in existing compiler optimizations, reducing the code size increase of our optimizations by up to 50% compared to not using the proposed cost model in these optimizations, without sacrificing performance. Our evaluation demonstrates that the cost model allows optimizations to perform fine-grained code size and performance trade-offs outperforming hard-coded heuristics.

[*]This research project is partially founded by Oracle Labs.

CCS Concepts • **Software and its engineering → Just-in-time compilers**; **Dynamic compilers**; *Virtual machines*;

Keywords Intermediate Representation, Cost Model, Performance Estimation, Just-In-Time Compilation

ACM Reference Format:
David Leopoldseder, Lukas Stadler, Manuel Rigger, Thomas Würthinger, and Hanspeter Mössenböck. 2018. A Cost Model for a Graph-Based Intermediate-Representation in a Dynamic Compiler. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '18), November 4, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3281287.3281290

1 Introduction

Optimizing compilers perform many optimizations on a compilation unit in order to improve the run-time performance of the generated machine code [2, 34]. However, there are optimizations that, while having a positive impact on one aspect (e.g., run-time performance), can have a negative impact on another (e.g., code size) [26, 34]. Such optimizations need to find a good compromise between benefits on metrics such as performance, with negative side effects on other metrics, such as code size. In order to estimate any effect on the quality of the produced code, such optimizations are typically based on abstract models of a compilation unit to reason about potential benefits and adverse impacts after applying a transformation [2]. These models are typically hand-crafted and highly specific to one particular optimization[1]. Thus, state-of-the-art compilers often contain multiple such models, up to one for each optimization applying a specific trade-off function. Changes to the structure of the compiler's IR or the ordering of optimizations as well as new optimizations and instruction types are rarely accounted for correctly in all trade-off functions, leading to unnecessary misclassifications of transformations. Additionally, over time, this increases maintenance costs.

[1]For example the inlining heuristics of the HotSpot [20] server compiler [32] in http://hg.openjdk.java.net/jdk10/hs/file/d85284ccd1bd/src/hotspot/share/opto/bytecodeInfo.cpp see `InlineTree::should_inline`.

Models have been devised for performance prediction in compilers for low-level languages that precisely estimate a program's performance at run time [35, 42][2]. In theory, those models could be used to guide dynamic and just-in-time compiler optimizations in their trade-off decisions. However, they cannot be applied to JIT compilation of high-level languages for several reasons, including:

- Dynamic compilers for high-level languages apply optimizations on abstraction levels where no notion of the architecture and platform is present in a compiler's intermediate representation. This prohibits the compiler from using precise performance prediction approaches that model all hardware and software peculiarities of a system.
- High-level optimizations are typically executed early in the compilation pipeline and the compiler can still significantly change the code by applying a pipeline of various high-level compiler optimizations [40] in between. This makes it difficult to predict the generated code that could be measured by a model designed for a low-level language.
- Precise performance prediction is often not suitable for JIT compilation as it is run-time-intensive.

Consequently, we believe that precise performance models [42] are not applicable to be used in high-level compiler optimizations. Optimizations often require cost models allowing them to compare intermediate representation (IR) instructions against each other on an abstract, relative level without reasoning about absolute code size or run-time metrics (e.g., cycles), yet still enabling an optimization to make valid assumptions about the relative run-time comparison of two code fragments.

In this paper, we present a cost model that we implemented for the IR of the Graal compiler. The cost model is designed to be simple to avoid complexity and over-specification of instructions. It enables Graal's duplication-based optimizations to make beneficial assumptions about code size and peak performance impact of a transformation on a compilation unit. We have already used the cost model in two duplication-based compiler optimizations: dominance-based duplication simulation [26] and fast-path loop unrolling of non-counted loops [25]. In this paper, we show how our cost model allowed for a straightforward implementation of a trade-off function in these optimizations. We further show how the cost model enabled us to reduce compilation time and code size without sacrificing peak performance in our optimizations. This allows the compiler to reduce compilation time and code size by factors up to $2x$ compared to a version without a cost-model-based trade-off. The cost model was developed over a period of nearly two years, and is now part of the Graal compiler. In summary, this paper contributes the following:

- We devised a cost model for a JIT compiler's IR that can be used in trade-off functions of compiler optimizations (Sections 2 and 3).
- We implemented the cost model in a Java bytecode-to-native-code just-in-time compiler to use it in two major optimizations that we previously presented (Sections 3 and 4).
- We evaluated those optimizations using our cost model on a set of industry-standard Java and JavaScript benchmarks (Section 5), showing significant improvements of performance, reduction of code size, and compilation time, while allowing for fine-grained trade-offs in these metrics.

2 System Overview

We implemented our IR cost model in the *Graal* compiler [31, 34, 45], a dynamically optimizing JIT compiler for *GraalVM* [31]. GraalVM is a modification of the *HotSpot* [20] Java virtual machine (JVM) where Graal replaces the *server compiler* [32] as HotSpot's top-tier compiler.

Java source programs are compiled to Java bytecode [27] by the `javac` compiler. This bytecode is loaded and interpreted upon invocation by the JVM. If certain code is considered important [20, 41], Graal is invoked with the bytecode of a method. It compiles the bytecode to machine code, after building an IR from it, which it uses to optimize the code during compilation. The IR undergoes several transformations: It starts as an abstract, platform-agnostic representation of the Java program, which is then gradually lowered to a platform-specific version from which machine code is generated. The compilation pipeline is split into a frontend and a backend. The frontend consists of three tiers: high-tier (being platform-agnostic and on an abstract Java level), mid-tier (being architecture-agnostic, but platform-specific) and low-tier (being both architecture and platform specific). The low tier's final IR is used to generate another low-level IR [22], on which Graal performs peephole optimizations [28], does register allocation [15, 43] and emits the final machine code. Graal has backends for x86-64, Sparc and aarch64. Graal's

```
interface F { int foo(); }
abstract class C implements F { int result; }
void foo(C c, int a) {
 int r;
 if (c.result > 0) r = a;
 else r = c.foo();
 c.result = r;
}
```

Listing 1. Trivial Java Program.

high-level IR [11–14] is a graph-based IR based on the *sea-of-nodes* IR [6, 7, 32] in static single assignment (SSA) [8] form. The IR is a super-position of two graphs: a control flow graph (downward pointing edges in Figure 1) and a data flow graph (upward pointing edges in Figure 1). Control flow nodes are

[2]LLVM [23] uses instruction costs in their vectorization cost model https://github.com/llvm-mirror/llvm/blob/master/include/llvm/Analysis/TargetTransformInfo.h line 134.

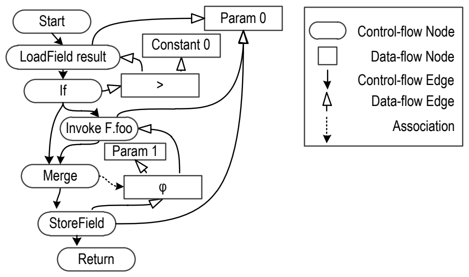


Figure 1. Graal Compiler IR after bytecode parsing.

fixed in their order, and seldom re-ordered by optimizations in the compiler. Data flow nodes can be freely re-ordered during compilation (i.e., they are free of side-effects). Their final position in the generated code purely depends on the scheduling [6] of their input dependencies.

Figure 1 shows Graal IR after bytecode parsing for the method from Listing 1. It is difficult even for such a small graph to estimate the performance and the size of the produced machine code. Method `foo` contains an interface call. The `invoke` node is lowered and then compiled to machine instructions that can take up between 4 and 80 bytes, as the compiler may need to generate code that searches for the dynamically-bound callee. However, code for the interface call is only generated if the compiler cannot devirtualize [21] the call and/or inline the method.

3 Node Cost Model

Compilers like Graal, which optimize high-level languages, could profit from abstract cost models to reason about optimization potentials instead of relying on hard-coded heuristics. This section presents the design of such a cost model and its implementation in the Graal compiler.

3.1 The Necessity of an IR Cost Model

Many optimizations in the Graal compiler are done when a program is still in a platform- and architecture-agnostic, high-level representation. Optimizations at this level leverage knowledge about the semantics of the JVM specification. Prominent examples of such optimizations are inlining [34], partial escape analysis [38] and dominance-based duplication simulation [26]. While high-level optimizations can significantly improve program performance, they often lack knowledge on how a given piece of IR is eventually lowered to machine code, especially when taking into account how subsequent compiler optimizations could transform it. Thus, compilers typically resort to heuristics when performing optimization decisions. While some optimization can significantly increase program performance, some entail negative effects. For example, code duplication [26], an optimization that sacrifices code size for performance, can lead to an exponential code explosion. A cost model must enable the compiler to perform deterministic estimations of code size on the IR level that correlates with the final machine code size to control code size expansion. We summarize these needs as *Problem 1 (P1)*.

P1 Machine Code Size Prediction High-level, platform- and architecture-agnostic, compiler optimizations cannot reason about system details which are necessary to predict which code will be generated for high-level language constructs, e.g., the code generated for the call instruction in Listing 1. Additionally, optimizations cannot infer the final machine code and thus not the size of a compilation unit, if there are other optimizations executed in between.

Compilers use different static and dynamic metrics extracted from a compilation unit to estimate code size. We summarize the metrics most common in current compilers in Table 1. Prior to our work, the predominant metric used in the Graal compiler was IR node count which was often sufficient; however, as outlined in Table 1, and verified by structured experiments, it can also lead to a misclassification of the code size of a method, mainly because of nodes that expand to multiple machine instructions and because of compiler intrinsics. Intrinsics are substitutions in the compiler to efficiently implement known methods of the standard library. For example, Graal implements intrinsics for array operations represented by a single instruction or a single node in the IR that later expands to the real operation, consisting of a large number of instructions inlined into the IR or even a call to a stub of code implementing the intrinsic.[3] Thus, we formulate *Problem 2 (P2)*.

P2 High-Level Instructions IR nodes capturing abstract language semantics and compiler intrinsics can expand to multiple instructions during code generation. Thus, they require special treatment when classifying code size in a compiler.

In addition to the above problems, we also consider structural canonicalizations as a source of misclassification, i.e., transformations that are applied by a compiler to bring a program into a canonical[4] form. Many compilers, including Graal, combine real optimizations like constant folding and strength reduction [2] with structural canonicalizations like always enforcing an order for constant inputs on expression nodes. If compiler optimizations (i.e., enabling optimizations like duplication) base their optimization decisions on the optimization potential of other optimization, and the potential is of structural nature, this can cause misclassification

[3] For example, in Graal, every method call to `Arrays.equals([] a,[] b)` in compiled code is intrinsified by the compiler with a special IR node expanding to more than 64 machine instructions (depending on the used CPU features).

[4] Canonical in a sense that it is understood by all optimizations.

Table 1. Compiler Code Size Quantification

Metric	Compilers using this Metric	Disadvantages
Bytecode Size	A metric used by many compilers mostly for inlining [1]. In HotSpot, the C1 [22] and C2 [32] compilers use it.	Bytecode is not optimized. This means that debugging code, assertions, logging, etc., and redundant computations that are easily eliminated with global value numbering [2] also account to code size. This can result in a miss-classify of code size.
IR Node Count	The C1, C2 and Graal compiler use(d) this metric.	IRs typically contain meta instructions / nodes [6, 11, 22] that are needed by the compiler to annotate certain instructions with additional information. However, such nodes often do not result in generated code, for example like Graal's nodes for partial escape analysis [38], which are required to represent virtualized objects. However, those nodes never generate code, they are only needed to support re-materialization during deoptimization. Additionally, there are large differences in the number of machine code bytes generated for different types of IR nodes. For example a field access typically lowers to a memory read of 4 byte size, whereas the invocation from Listing 1 can take up to 80 bytes of machine code. Notably, both operations are represented with one node and thus count as 1.

of the optimization potential. This is the case as an optimization might perform a transformation because it enables a structural canonicalization that will not cause a performance benefit. This forces optimizations to understand what a performance-increasing code transformation is and what is not (*Problem 3 (P3)*).

P3 Performance Estimation Compiler optimizations typically cannot infer the performance increase of a particular transformation.

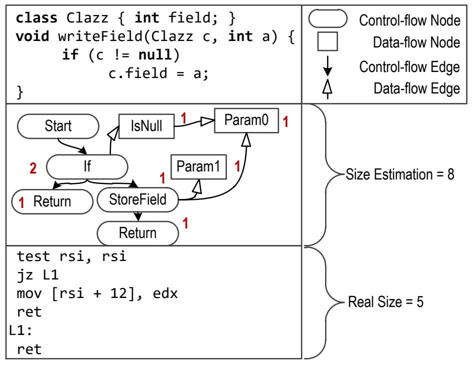
In order to by used by Graal the cost model must be applicable to JIT compilation, where the worst case asymptotic complexity should be at most linear over the number of nodes.

3.2 Node Cost Model

In order to solve the presented problems, we propose the introduction of an IR cost model for the Graal compiler entitled *node cost model* that specifies costs for IR nodes. These costs are abstract metrics for code size (`NodeSize`) and operation latency (`NodeCycles`). Costs are specified together with the definition of an IR node.

`NodeSize` is a relative metric that we define as the average number of generated instructions on an abstract architecture after lowering the operation to machine code. We base the definition roughly on a double word instruction (32 bit) format without modeling different architectures. Our assumption is that an addition always generates one instruction of machine code, irrelevant of the abstraction level. The addition operation is our baseline. We specify other operations relative to it. We have a loose definition in order to avoid over-specification. For high level optimizations it is not important to have byte or word precision.

`NodeCycle` is an estimation of the latency of an instruction relative to an addition operation. We assume the latency of an addition is one cycle on every platform and specify all other instruction latencies relative to an addition operation. The same rules as for the `NodeSize` metric apply. If a high-level operation (e.g., a field access) can expand to multiple instructions being executed, we assume the most likely path through the compilation pipeline and specify the cycles metric as an average of the latency of the operation. We gathered the values for the latencies of instructions relative to the addition from several sources mostly by empirical evaluation and by instruction classification approaches [16].



```
class Clazz { int field; }
void writeField(Clazz c, int a) {
    if (c != null)
        c.field = a;
}
```

```
 test rsi, rsi
 jz L1
 mov [rsi + 12], edx
 ret
L1:
 ret
```

Figure 2. Node Cost Model Code Size Estimation: Red numbers represent `NodeSize` values. We removed HotSpot-specific prologue and epilogue instructions for simplicity.

Code Size Estimation The cost model can be used to solve *P1* from Section 3.1 by performing code size estimations in optimizations. The compiler can use these estimates in duplication-based optimizations to trade-off between several optimization candidates. Consider Figure 2, which shows a simple Java method with the associated Graal IR together with the resulting assembly code in *x86-64* Intel assembly. The compiler overestimated the `NodeSize` for the method `writeField` to be 8 machine instructions due to two reasons: (1) It pessimistically assumes the costs of accessing a parameter when it resides on the stack; however, in this case the parameters can be accessed in registers due to the calling convention. (2) The compiler assumes that an `If` node generates 2 jump instructions, namely one from the true-branch to the merge and one from the comparison to the false branch; however, if both branches end in control flow sinks and never merge again one jump is saved.

Relative Performance Prediction The second major use

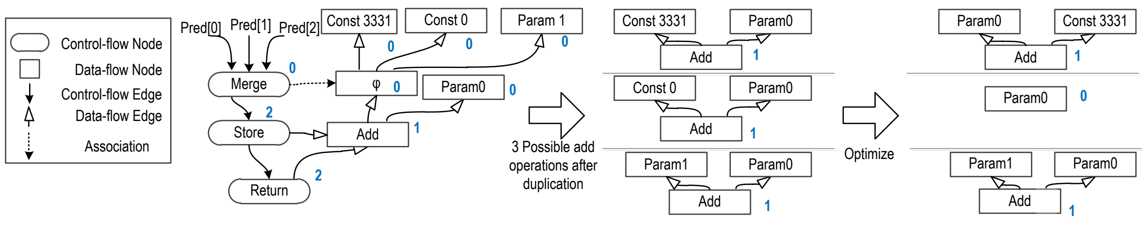


Figure 3. Node Cost Model Cycles Estimation Use Case: Blue numbers represent NodeCycles values.

case of the model is relative performance prediction of optimization candidates solving *P3* from Section 3.1. The node cost model can be used to compare two pieces of code based on their relative performance estimation. This need arose during our work on code duplication. Many optimization patterns are hard-coded in compilers: this includes optimizations like algebraic simplifications and strength reductions. However, simplifications do not guarantee peak performance increases. Therefore, comparing two pieces of code by summing up their relative performance predictions normalized to their profiled execution frequency allows compilers to perform better performance trade-offs.

Consider the example in Figure 3 that shows Graal IR for a control flow merge with three predecessors. In SSA, a variable is only assigned once; to model multiple possible assignments, φ functions are used. Figure 3 contains one φ at the merge with 3 inputs. Code duplication tries to determine how the code on each path looks after duplication, in order to find the best candidate for optimizations. If `pred[0]` is duplicated, the resulting addition operation with a constant and a parameter cannot be further optimized. However, Graal would still swap the two inputs to have a canonical representation of arithmetic operations for global value numbering. This is not a performance-increasing optimization, but a structural canonicalization (see *P3* from Section 3.1). Graal combines such transformations with actual transformations, making it impractical for optimizations to know the difference if not hard coded. In this case, the structural canonicalization will not result in a performance gain. Duplicating `pred[1]`, however, would lead to an addition with 0 that can be folded away completely. This requires the optimization to understand what the real benefit of a transformation is in terms of performance. Using the node cost model, the compiler can see, precisely, that only the duplication of `pred[1]` can generate a real improvement in performance. This is important as the store and return instructions are duplicated as well which increases the code size. The compiler has to trade-off the benefit with the code size, and in this example, only the duplication of `pred[1]` generates a sufficient improvement to justify the increase of code size.

Discussion The node cost model is not a precise performance prediction framework. We do not model pipelined architectures as we have no notion of the concrete hardware during our high-level optimizations. Therefore, we also make no distinction between throughput and latency. We avoid modeling out-of-order processors, cache misses, data misalignments and exceptions in our cost model. We assume a single threaded in-order execution environment with perfectly cached memory accesses. This is desired as Graal is a multi-platform compiler and modeling a specific hardware including all peculiarities would involve much implementation and compile-time effort.[5] We assume that a precise estimation is not necessary to support compiler optimizations in their trade-off decisions. This claim is verified in Section 5. For code size specifications, we also explicitly omit modeling debug information that is emitted by the code generator. Some instructions, for example implicit null checks, require the compiler to generate debug information for the VM during code generation. We also ignore register allocation effects by assuming an infinite number of virtual registers.

3.3 Implementation

We implemented `NodeSize` and `NodeCycles` for over 430 nodes in the Graal compiler. Graal IR nodes are specified as extensible Java classes. Classes representing nodes in the IR must be annotated with a special `NodeInfo` annotation to which we added additional fields for an abstract estimation of code size and run-time latency (see Listing 2). Figure 4

```
public @interface NodeInfo {
 NodeCycles cycles() default CYCLES_UNSET;
 NodeSize size() default SIZE_UNSET;
}
```

Listing 2. Excerpt of the NodeInfo Annotation.

shows the distribution of values for `NodeSize` and `NodeCycles` in the Graal compiler. We can see that a significant amount of nodes are ignored in the cost model as their values do not generate code or do not consume CPU cycles (i.e., they are assigned the value `IGNORED`). Those nodes would be a source for misclassification of optimizations as they are noise to the real value distributions of the code. Additionally,

[5] Our implementation allows for architecture specific `NodeSize` and `NodeCycle` values if a particular architecture has wildly different characteristics. However, we have not implemented architecture specific characteristics currently.

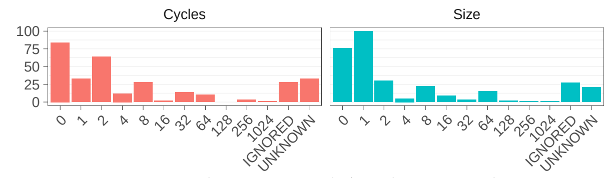


Figure 4. Node Cost Model Value Distributions

we solved *P2* from Section 3.1 by also annotating IR nodes representing intrinsic operations with a code size and latency estimation (if possible). However, two problems remain for the cost model properties. The first problem is that there are nodes for which we cannot make a static estimation. For example, Graal supports grouping allocations of multiple objects together. This is expressed as a group allocation node whose latency is dependent on the number of allocations and the allocation sizes, thus the latency has to be inferred dynamically. The second problem is that there are nodes for which we simply cannot make an estimation as their code size / latency is completely unknown. There are two kinds of nodes for which we cannot predict latency at compile time:

Call Nodes The latency of invocations is opaque to the compiler. The compiler would need to parse every possible callable method to estimate its run time. Thus, we ignore the latency of calls in the cost model. We annotated call nodes with estimations for their `NodeSize` but the run-time latency is set to `UNKOWN`.

Deoptimization Nodes Deoptimization [19] causes the program to continue execution in the interpreter and is thus out of scope of any performance prediction.

The concept of unknown values is a potential threat to misclassification of code. If optimizations make decisions based on control paths containing `UNKNOWN` nodes those decisions are potentially based on a misclassification of size or latency. However, in practice, such cases do not happen often. Deoptimizations are rare and are treated as control flow sinks[6] for all optimizations (i.e., a path leading to a deoptimization should not be optimized aggressively) and the compiler has an estimation for the code size of a deoptimization. Invokes are similar: the compiler can infer the generated code for an invocation, but the latency is unknown. Invokes themselves can have arbitrary side effects, so they can never be removed by the compiler. The latency estimation for a caller remains the same if we exclude the latency of the callee from the computation. The latency of a callee can be treated as a constant, scaling the latency of the caller and is thus irrelevant for the performance estimation of the caller.

Table 2 shows an excerpt of important nodes in the Graal compiler together with their cost model values. The addition node is the baseline for the rest of the instructions. For example, multiplication typically needs twice the latency of an addition. `LoadField` nodes access memory and thus we

[6]Nodes in the IR that end a particular control path.

Table 2. Important Node Cost Model Nodes

Node	`NodeCycles`	`NodeSize`
AddNode	1	1
MulNode	2	1
LoadField	2	1
DivNode	32	1
Call (Static Call)	Unkown	2

estimate the time needed to `mov` a value from cached memory to a register. A field access in Java also requires a null check. This check can be done implicitly or explicitly. However, after bytecode parsing, the compiler does not know yet if it can emit an implicit null check, thus it estimates the latency to be 2 cycles, as an explicit null check typically requires 1 instruction and 1 cycle. Integer division (`div`) is an instruction where the latency depends on the size of the operands and the value ranges, thus we take the worst-case latency estimation of the instruction as a reference value. Static calls spend an `UNKNOWN` amount of cycles in the callee, but typically take up to two machine instructions.

4 Node-Cost-Model-based Optimizations

Several optimizations in the Graal compiler use the proposed node cost model, the two most important ones being dominance-based duplication simulation [24, 26] and fast-path unrolling for non-counted loops [25]. This section illustrates how they utilize the node cost model in their trade-off functions. Note that also other optimizations in the Graal compiler such as floating-node duplication and conditional move optimization use the node cost model in a similar way.

Dominance-based Duplication Simulation (DBDS) [26] is a code duplication optimization that hoists code parts from control flow merge blocks into predecessor blocks in order to specialize them to the incoming branches. DBDS uses the cost model in two ways: (1) It estimates the code size increase of a duplication by summing up the `NodeSize` for all nodes in a merge block, (2) and it estimates the run-time improvement of each duplication by simulating a duplication and computing the difference in `NodeCycles` to the merge block before duplication. This way benefit and cost are computed for each possible duplication and used in a trade-off heuristic to find beneficial optimization candidates. The trade-off is based on the ratio between estimated performance increase and code size increase. Below we include the important parts of the trade-off heuristic that we previously described [26]:

$$c \ldots \text{Cost} \equiv \text{Code Size Increase}$$
$$b \ldots \text{Benefit} \equiv \text{Cycles Saved} \times \text{Execution Probability}$$
$$\texttt{shouldDuplicate} \mapsto b > c$$

Before we had used the node cost model, DBDS worked with hard-coded optimization patterns not covering all performance increases presented in [26]. Section 5 shows the performance of DBDS with a cost model and without it, showing a significant improvement when using the node cost model.

Fast-Path Unrolling of Non-Counted Loops [25] is an optimization for non-counted loops [9].[7] It attempts to improve performance by estimating the optimization potential of a loop after unrolling by simulating the unrolling of (multiple) iterations of the loop. The estimated performance improvement of possible unrolling transformation is then compared with the code size increase and a trade-off function determines if it is beneficial to unroll a given loop. The key components of the trade-off heuristic that we presented in [25] are shown below:

$$\begin{aligned} s &\ldots \text{Loop Size} \\ cp &\ldots \text{Cycles saved per loop iteration} \\ bs &\ldots \text{Bytes per Cycle Spending} \\ \texttt{shouldUnroll(loop)} &\mapsto\ cp * bs > s \end{aligned}$$

The benefit is expressed as the number of estimated `NodeCycles` saved per loop iteration after unrolling a loop. This benefit is compared with the size of the loop body, estimated from the `NodeSize` sum of the loop. The body is duplicated once if the loop is unrolled once. Section 5 shows how fast-path unrolling can improve the performance of e.g. the *jetstream* benchmarks.

5 Evaluation

In our evaluation, we want to demonstrate the effectiveness of the proposed node cost model.

Hypothesis We want to show that our node cost model allows optimizations to apply a finer grained trade-off of code size versus peak performance, resulting in increased peak performance and reduced code size and compilation time. We tested this hypothesis by running a set of industry-standard Java and JavaScript benchmarks with different compiler and cost model configurations.

Benchmarks We evaluated the performance of our node cost model on the *Java DaCapo* [3] [8], *Scala DaCapo* [30, 34, 36, 37] , *JavaScript Octane* [5] and *Jetstream* [33][9] benchmarks. The Java and Scala benchmarks were directly executed on GraalVM whereas the JavaScript benchmarks were executed on Truffle [44, 45], the partial-evaluation [17] framework on top of the Graal compiler.

Environment We executed all benchmarks on an Intel© i7-6820HQ machine running at 2.7GHz, equipped with 32 GB DDR-4 RAM. We disabled frequency scaling and turbo boost to obtain more stable results. For each benchmark, we measured performance (run time or throughput) of the application, code size and compilation time of the Graal compiler.

Configurations Given our initial hypothesis, a change in the cost model parameterization should be reflected in either performance, code size, or compile time. Therefore, we tested five configurations and parameterizations of the node cost model:

- **`no-opt`** Duplication-based optimizations from Section 4 are disabled, but otherwise the GraalVM standard configuration is used. We use this configuration as the baseline for all our experiments and normalize the other configurations to it.
- **`GraalVM`** The default configuration of GraalVM, which is configured to deliver high performance at a medium code size and compile time increase. It uses the node cost model in duplication-based optimizations to perform code size and run time estimations.
- **`no-model`** The cost model trade-off functions for DBDS and fast-path unrolling are disabled. Every time an optimization finds a potential improvement in peak performance, it performs the associated transformation without a trade-off against code size.
- **`zeroSize`** In this configuration, we set all `NodeSize` values for IR nodes to 0. The `zeroSize` configuration should behave similarly to the `no-model` configuration because the estimated code size increase is always 0. However, the `GraalVM` configuration limits the maximum code size increase per compilation unit to 50%, which will also be used in the `zeroSize` configuration. Therefore, we expect code sizes larger than `GraalVM` but smaller than 50%.
- **`zeroCycles`** In this configuration, we set all `NodeCycles` values for IR nodes to 0. The `zeroCycles` configuration should reduce the code size compared to the `no-opt` configuration because the trade-off function of DBDS from Section 4 only triggers if the code size effect of a duplication is negative (i.e., duplication can enable subsequent compiler optimizations that reduce code size like, e.g., dead code elimination [2]).

Expected changes in performance for all configurations in relation to `no-opt` can be seen in Table 3.

Table 3. Expected Performance Impacts: Upward pointing arrows represent an increase of a given metric, downward pointing arrows a decrease.

Config	Performance	Code Size	Compile Time	Relative to
`no-opt`	-	-	-	-
`GraalVM`	↑	↑	↑	`no-opt`
`no-model`	↑	↑	↑	`GraalVM`
`zeroCycles`	-	↓	↓	`no-opt`
`zeroSize`	-	↑	↑	`GraalVM`

[7]Non-counted loops are loops for which we cannot statically reason about induction variables and iteration count of a loop.
[8]We excluded the benchmarks eclipse, tomcat, tradebeans and tradesoap as our version of the DaCapo benchmarks no longer supports them with JDK versions > 8u92.
[9] We excluded all non asm.js [29] benchmarks from the jetstream suite as a large part of the jetstream benchmarks have been previously published in the octane suite.

For each configuration, we executed a benchmark 10 times and computed the arithmetic mean from the last n iterations of the benchmark after compilation frequency stabilizes; this effectively removes warm-up from the results. The number of measurement iterations n is computed from the maximum number of benchmark iterations and is at least 20% of it (effectively between 5 and 10).

Synthesis The results of the experiments can be seen in the boxplots in Figures 5a to 5d. For each benchmark, we plotted compile time, code size and performance (run time or throughput). We normalized the results to a mean computed from the `no-opt` configuration, without any node-cost-model-based optimization enabled. The results seem to confirm our hypothesis. The `GraalVM` configuration produces the best performance at a medium code size and compile time increase. For some benchmarks, `GraalVM` produces less optimal code than `no-opt`; however, this only applies to outliers (for details about DBDS performance see [26]). The interesting comparison is between `GraalVM`, `no-model`, `zeroSize` and `zeroCycles`. The `GraalVM` configuration using the node cost model to do relative performance estimations and code size classification always results in less code size and faster compilation times than the `no-model` configuration. The `zeroCycles` estimation indeed either produces no code size increase or reduces the code size compared to the `no-opt` configuration. In benchmarks like *zlib*, `zeroCycles` results in less compile time than `no-opt` as duplication can significantly reduce code size, resulting in less work for other compiler optimizations. The `zeroSize` configuration also behaves as expected by producing code size increases and compile time increases between `GraalVM` and `no-model`.

The results indicate that the cost model is superior to hard-coded heuristics in the two presented optimizations. The most interesting configuration is the `zeroCycles` one, as it shows that a fine-grained trade-off is possible even with a relative cost model. In `zeroCycles`, each IR node has a `NodeCycle` value of 0. As shown in Section 4, DBDS only duplicates code iff benefit $*$ probability $>$ cost. In this configuration, the left hand of the inequality is always 0, because the $benefit$ is 0 (as all `NodeCycles` are 0). Thus, DBDS only duplicates if the *cost* is negative, which can happen when the impact of a duplication is an enabled dead code elimination resulting in a reduction of code size. This effectively leads to a reduction in code size compared to the `no-opt` configuration in some benchmarks. The `zeroSize` configuration behaves like the `no-model` configuration except that it implements the upper code size boundary of the `GraalVM` configuration for an entire compiler graph, and thus produces less code than the `no-model` configuration.

The results show that a simple cost model capturing relative instruction latencies and instruction sizes can be used in high-level optimizations to guide trade-off functions. This enabled us to implement the trade-off function in our duplication optimizations in Graal [25, 26]. Without investing significant amounts of compile time we could significantly improve optimizations to perform fine-grained performance predictions at an abstract, architecture-agnostic, optimization level. This allowed us to improve performance and to reduce code size and compile time.

6 Related Work

Various research fields applied cost models to reduce compilation times and code size, to increase or predict performance or to select the best candidate instructions during code generation.

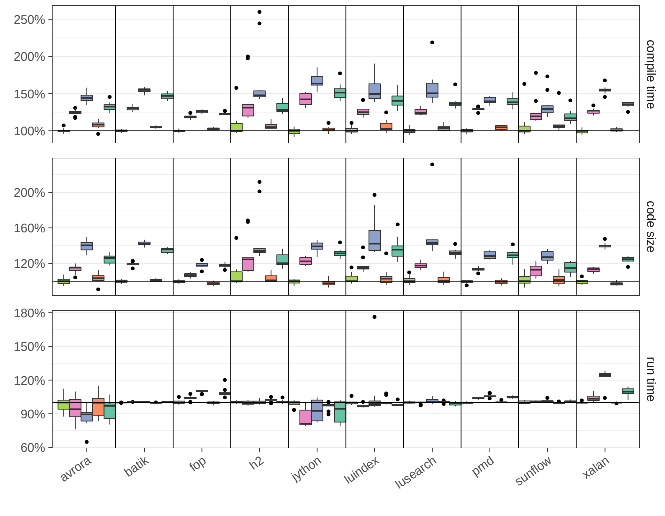
Production Compilers: Most industry-standard compilers (e.g. LLVM[23] and GCC [18]) use cost models to implement performance predictions. However, they typically use architecture-specific information in their models. We instead propose an architecture-agnostic cost model for high-level optimizations.

(Precise) Performance Prediction: Research in performance estimation explores ways to estimate the run time of an application without executing it. Various approaches have been investigated including precise performance prediction in compilers [4, 42], compile-time prediction of single-to-multicore application migration [39], compile-time estimations for speculative parallelization [10], and execution time derivations for benchmark characterization to estimate the performance of a program on a given hardware [35]. Performance prediction approaches are typically done for compilers of static languages, enabling a precise analysis and estimation, assuming a late stage in the compilation pipeline. Graal optimizations cannot make performance-relevant transformation decisions based upon IR that must not change any more until code generation. This prohibits modeling architecture-specific contexts, i.e., it effectively prohibits the use of such models in Graal.

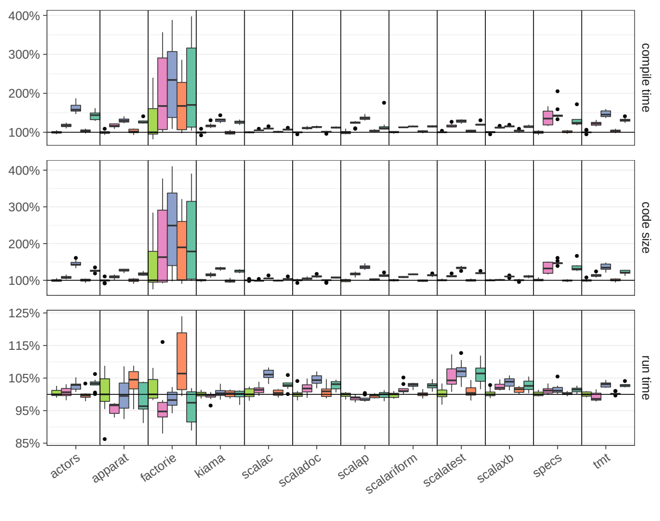
Instruction Scheduling & Selection: Other approaches use instruction cost models to find minimum-cost instruction sequences during code generation (e.g., in the HotSpot server compiler [32]). We improve upon this idea by using relative instruction latencies and size estimations for abstract, high-level, compiler optimizations.

7 Conclusion & Future Work

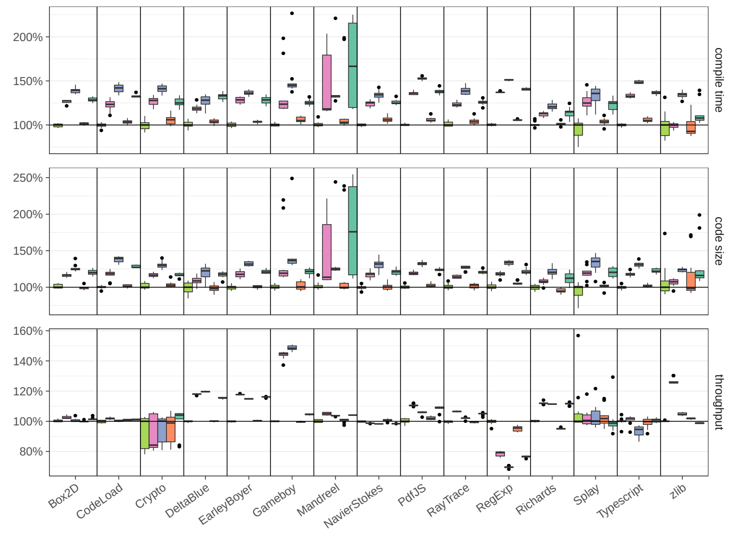
In this paper, we have presented a cost model for the IR of a dynamic compiler, which is used by several optimizations. We implemented the cost model in the Graal compiler and showed that it can be used to guide high-level optimizations to selectively control their effects on peak performance and code size. These findings imply that compilers like Graal do not need to implement complex, architecture-dependent cost models for all optimizations, but, for some optimizations, can resort to simpler, abstract cost models.



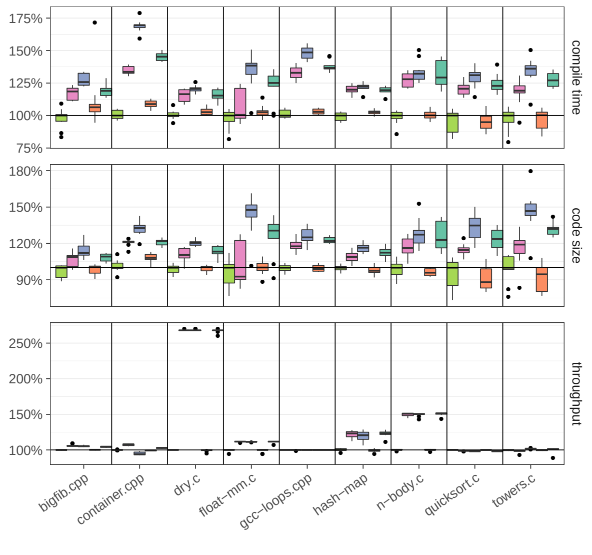
(a) Java DaCapo



(b) Scala DaCapo



(c) JavaScript Octane



(d) JavaScript jetstream

🟩 no−opt 🟪 GraalVM 🟦 no−model 🟧 zeroCycles 🟩 zeroSize

Figure 5. Performance Numbers: Compile time (lower is better), code size (lower is better), run time(lower is better), throughput (higher is better).

As part of our future work, we want to explore whether our cost model can be used in other optimizations of the Graal compiler. If so, we want to migrate them so that they benefit from our cost model.

References

[1] Matthew Arnold, Stephen Fink, Vivek Sarkar, and Peter F. Sweeney. 2000. A Comparative Study of Static and Profile-based Heuristics for Inlining. In *DYNAMO'00*. ACM, New York, NY, USA, 52–64. https://doi.org/10.1145/351397.351416

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

[3] S. M. Blackburn, R. Garner, et al. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA'06*. ACM Press, 169–190. https://doi.org/10.1145/1167473.1167488

[4] Calin Cascaval, Luiz DeRose, David A Padua, and Daniel A Reed. 1999. Compile-time based performance prediction. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 365–379. https://doi.org/10.1007/3-540-44905-1_23

[5] Stefano Cazzulani. 2012. Octane: The JavaScript benchmark suite for the modern web. *Retrieved December* 21 (2012), 2015.

[6] Cliff Click. 1995. Global Code Motion/Global Value Numbering. In *PLDI'95*. ACM, New York, NY, USA, 246–257. https://doi.org/10.1145/207110.207154

[7] Cliff Click and Keith D. Cooper. 1995. Combining Analyses, Combining Optimizations. *ACM Trans. Program. Lang. Syst.* 17, 2 (March 1995), 16. http://doi.acm.org/10.1145/201059.201061

[8] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (Oct. 1991), 40. https://doi.org/10.1145/115372.115320

[9] Jack W Davidson and Sanjay Jinturkar. 1995. *An aggressive approach to loop unrolling.* Technical Report. Technical Report CS-95-26, Department of Computer Science, University of Virginia, Charlottesville.

[10] Jialin Dou and Marcelo Cintra. 2007. A Compiler Cost Model for Speculative Parallelization. *ACM TACO* 4, 2, Article 12 (June 2007). https://doi.org/10.1145/1250727.1250732

[11] 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.*

[12] Gilles Duboscq, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Speculation Without Regret: Reducing Deoptimization Metadata in the Graal Compiler. In *PPPJ '14.* ACM, New York, NY, USA, 7. https://doi.org/10.1145/2647508.2647521

[13] 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 *VMIL'13.* https://doi.org/10.1145/2542142.2542143

[14] Gilles Marie Duboscq. 2016. *Combining Speculative Optimizations with Flexible Scheduling of Side-effects.* Ph.D. Dissertation. Linz, Upper Austria, Austria.

[15] Josef Eisl, Matthias Grimmer, Doug Simon, Thomas Würthinger, and Hanspeter Mössenböck. 2016. Trace-based Register Allocation in a JIT Compiler. In *PPPJ '16.* ACM, New York, NY, USA. https://doi.org/10.1145/2972206.2972211

[16] Agner Fog. 2018. Instruction Tables for Intel, AMD and VIA CPUs. (2018). http://www.agner.org/optimize/instruction_tables.pdf

[17] Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation* 12, 4 (01 Dec 1999), 381–391. https://doi.org/10.1023/A:1010095604496

[18] GNU. 2018. GCC, the GNU Compiler Collection. (2018). https://gcc.gnu.org/

[19] Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging Optimized Code with Dynamic Deoptimization. In *PLDI'92.* ACM, New York, NY, USA, 32–43. https://doi.org/10.1145/143095.143114

[20] HotSpot JVM 2018. Java Version History (J2SE 1.3). (2018). http://en.wikipedia.org/wiki/Java_version_history

[21] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. 2000. A Study of Devirtualization Techniques for a Java Just-In-Time Compiler. In *OOPSLA '00.* ACM, 294–310. https://doi.org/10.1145/353171.353191

[22] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. 2008. Design of the Java HotSpot client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization (TACO)* 5, 1 (2008), 7.

[23] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO '04 (CGO '04).* IEEE Computer Society. http://dl.acm.org/citation.cfm?id=977395.977673

[24] David Leopoldseder. 2017. Simulation-based Code Duplication for Enhancing Compiler Optimizations. In *SPLASH Companion 2017.* ACM, New York, NY, USA, 10–12. https://doi.org/10.1145/3135932.3135935

[25] 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 *ManLang 2018.* ACM, New York, NY, USA. https://doi.org/10.1145/3237009.3237013

[26] 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 *CGO 2018.* ACM, New York, NY, USA, 126–137. https://doi.org/10.1145/3168811

[27] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2015. *The Java Virtual Machine Specification, Java SE 8 Edition.* http://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf

[28] W. M. McKeeman. 1965. Peephole Optimization. *Commun. ACM* 8, 7 (July 1965), 443–444. https://doi.org/10.1145/364995.365000

[29] Mozilla. 2018. asm.js. (2018). http://http://asmjs.org/

[30] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. 2004. *An overview of the Scala programming language.* Technical Report.

[31] OpenJDK 2017. GraalVM -New JIT Compiler and Polyglot Runtime for the JVM;. (2017). http://www.oracle.com/technetwork/oracle-labs/program-languages/overview/index-2301583.html

[32] Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java HotSpot™ Server Compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium.* USENIX, 1–12.

[33] Filip Pizlo. 2014. JetStream Benchmark Suite. (2014). http://browserbench.org/JetStream/

[34] Aleksandar Prokopec, David Leopoldseder, Gilles Duboscq, and Thomas Würthinger. 2017. Making Collection Operations Optimal with Aggressive JIT Compilation. In *SCALA 2017.* ACM, New York, NY, USA, 29–40. https://doi.org/10.1145/3136000.3136002

[35] Rafael H Saavedra and Alan J Smith. 1996. Analysis of benchmark characteristics and benchmark performance prediction. *ACM Transactions on Computer Systems (TOCS)* 14, 4 (1996), 344–384.

[36] 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 *OOPSLA '11.* ACM, New York, NY, USA. https://doi.org/10.1145/2048066.2048118

[37] Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck, Thomas Würthinger, and Doug Simon. 2013. An Experimental Study of the Influence of Dynamic Compiler Optimizations on Scala Performance. In *SCALA '13.* ACM, New York, NY, USA, Article 9, 8 pages. https://doi.org/10.1145/2489837.2489846

[38] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. [n. d.]. Partial Escape Analysis and Scalar Replacement for Java. In *CGO'14.* ACM Press, 165–174. https://doi.org/10.1145/2544137.2544157

[39] Munara Tolubaeva. 2014. *Compiler Cost Model for Multicore Architectures.* Ph.D. Dissertation.

[40] Sid-Ahmed-Ali Touati and Denis Barthou. 2006. On the Decidability of Phase Ordering Problem in Optimizing Compilation. In *CF '06.* ACM, 10. https://doi.org/10.1145/1128022.1128042

[41] April W. Wade, Prasad A. Kulkarni, and Michael R. Jantz. 2017. AOT vs. JIT: Impact of Profile Data on Code Quality. In *LCTES 2017.* ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/3078633.3081037

[42] Ko-Yang Wang. 1994. Precise Compile-time Performance Prediction for Superscalar-based Computers. In *PLDI'94.* ACM, New York, NY, USA, 73–84. https://doi.org/10.1145/178243.178250

[43] Christian Wimmer and Michael Franz. 2010. Linear Scan Register Allocation on SSA Form. In *CGO '10.* ACM, New York, NY, USA, 170–179. https://doi.org/10.1145/1772954.1772979

[44] 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 *PLDI 2017.* ACM, New York, NY, USA, 662–676. https://doi.org/10.1145/3062341.3062381

[45] 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 *Onward! 2013.* ACM, New York, NY, USA, 187–204. https://doi.org/10.1145/2509578.2509581