Taking a Closer Look: An Outlier-Driven Approach to Compilation-Time Optimization

Florian Huemer
Johannes Kepler University, Linz, Austria

David Leopoldseder
Oracle Labs, Vienna, Austria

Aleksandar Prokopec
Oracle Labs, Zurich, Switzerland

Raphael Mosaner
Oracle Labs, Linz, Austria

Hanspeter Mössenböck
Johannes Kepler University, Linz, Austria

Abstract

Improving compilation time in optimizing compilers is challenging due to their large number of interconnected components. This includes compiler optimizations, compiler tiers, heuristics, and profiling information. Despite this complexity, research in compilation-time optimization is often guided by analyzing metrics of entire program runs, such as the total compilation time and overall memory footprint. This coarse-grained perspective hides relevant information, such as source program functions for which the compiler allocates a lot of memory or compiler optimizations with a high impact on the total compilation time. This leaves high-level metrics as the only reference point for driving optimization design. Consequently, compilation-time regressions in one program function that are obscured by improvements in other functions stay undetected, while the impacts of compiler changes on untouched parts of the compiler are mainly unknown. Furthermore, developers overlook long-standing compiler defects because their high-level metrics do not change over time.

To address these limitations, we propose ICON, a new data-driven approach to compilation-time optimization that breaks up high-level metrics into individual source program functions, compiler optimizations, or even into individual instructions in the compiler source code. Our methodology enables an iterative in-depth compilation-time analysis, focusing on outliers to identify optimization opportunities. We show that outliers, both in terms of time spent in a particular compiler optimization, and in terms of individual compilations that take substantially longer, can reveal potential problems in the compiler implementation. We applied our approach to GraalVM and extracted data for multiple of its language runtimes. We analyzed the resulting data, present the first detailed look into the distribution of compilation time in the GraalVM compiler, a state-of-the-art multi-language compiler, and identified defects that led to regressions in overall compilation time or the compilation time of specific languages. We furthermore designed two optimizations based on the identified outliers that improve compilation time between 2.25% and 9.45%. We believe that our approach can guide compiler developers in finding usually overlooked optimization potential and defects, and focus future research efforts in making compilers more efficient.

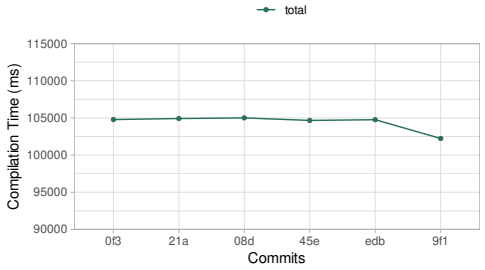
2012 ACM Subject Classification General and reference → Performance; General and reference → Measurement; Software and its engineering → Just-in-time compilers; Software and its engineering → Dynamic compilers

Keywords and phrases Compilation time, outliers, dynamic languages, virtual machines, GraalVM, ICON

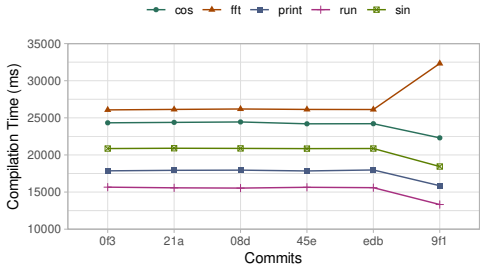
Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.20

38th European Conference on Object-Oriented Programming (ECOOP 2024).
Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 20; pp. 20:1–20:28

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



(a) Results of a hypothetical benchmark measuring total compilation time over the course of several commits. Lower is better.



(b) Results split into individual function compilation times over the course of several commits. Lower is better.

1 Introduction

Improving compilation time in optimizing compilers is challenging due to their large number of interconnected components. Multiple interacting compiler optimizations [24], several compiler tiers, various heuristics, and different profiling information make it challenging to identify the impact of compiler source code changes on the total compilation time of a program. Compilation time should be minimal, especially in modern cloud applications based on serverless functions or microservice architectures that are executed on demand and started frequently [42, 34]. The compilation time for these services is even more relevant for runtimes using just-in-time compilation, where the compilation time directly impacts the startup time of a service [46]. In these runtimes, savings in compilation time are not limited to a single compilation upfront but are reapplied every time an application is started and recompiled. Ahead-of-time-compiled runtimes should also strive for minimal compilation time such as in continuous integration and continuous delivery infrastructures, where compilations take up a large portion of the resource utilization as part of build steps [3].

Despite the high complexity of optimizing compilers and a high demand for fast compilation, research in this field often relies on metrics reflecting entire program runs, such as total compilation time or overall memory footprint, to guide compilation-time optimization. Using fine-grained metrics, such as the time spent compiling individual program functions or the time spent in specific compiler optimizations, could reveal outliers in compilation time that lead to defects and optimization opportunities in the compiler.

To illustrate this problem, consider the hypothetical implementation of a new compiler phase `p` that introduces an optimization to a compiler. To evaluate the compilation-time impact of `p`, the conventional approach of analyzing the total compilation time of several benchmarks and comparing it with previous records seems appropriate. As shown in Figure 1a, which illustrates the total compilation time of a hypothetical benchmark over the course of several commits, `p`, implemented in `9f1`, leads to a compilation-time decrease, leaving the impression of a positive compilation-time impact. While this impression seems correct overall, it hides valuable information which is only unveiled by analyzing the compilation time of individual benchmark functions. As shown in Figure 1b, `p` improves the compilation time of nearly all functions in the benchmark but also results in an outlier with a significant increase in compilation time, the `fft` function. This outlier points towards a defect in `p` that only occurs under certain conditions, which seem to be present in the `fft` function, and requires further investigation. Consequently, relying solely on metrics of entire program runs to evaluate compilation-time performance can obscure newly-introduced or long-standing defects in compiler implementations.

Therefore, our contribution is to describe a new approach to compiler optimization focusing on outliers to identify defects:

- We describe *Iterative Compilation-time optimization through Outlier-driven Narrowing* (ICON), a novel data-driven approach to compilation-time optimization that splits compilation metrics into individual functions, compiler optimizations, or even into individual instructions in the compiler source code to identify potential problems in compiler implementations by focusing on the outliers in extracted data (Section 3). The approach combines a fine-grained metrics extraction based on iterative narrowing with an outlier-focused approach to finding potential optimization opportunities.
- We present the first detailed look into the distribution of compilation time in the GraalVM compiler, a state-of-the-art multi-language compiler (Section 4). We base our data on the evaluation of 94 benchmarks from the "Are We Fast Yet?"[1] [27], JetStream 2[2], "Computer Language Benchmark Game,"[3] [27] and several internal benchmark suites, analyzing compilation-time metrics for five runtimes, including Python and JavaScript.
- To demonstrate the effectiveness of the ICON approach, we conducted an outlier analysis on the GraalVM compilation-time metrics, identifying one language-agnostic and three language-specific outliers in compilation time (Section 4.5). Through the iterative application of our approach, we narrowed the scopes of the outliers in the compiler and discovered four defects in the GraalVM compiler that were responsible for sub-optimal compilation time in compiler optimizations.
- We sketch the design of two optimizations that target two of the defects identified by our outlier analysis and improve compilation time between 2.25% in Python and 9.45% in Java (Section 5).

2 The Environment of Our Study

We describe ICON as a general methodology to optimize the compilation time of compilers and apply it to a specific runtime environment to emphasize its applicability. We thus begin our technical content by describing GraalVM, the runtime we worked with.

2.1 GraalVM

GraalVM [45] is a state-of-the-art high-performance Java Virtual Machine (JVM) [26] that includes a dynamic optimizing compiler called *GraalVM compiler*. GraalVM provides native support for JVM languages, is implemented in Java [17], and supports just-in-time (JIT) and ahead-of-time (AOT) compilation through the *JVM* [45] and *Native-Image* [42] deployments. The JVM deployment starts programs in the Java interpreter and just-in-time compiles frequently executed methods with the GraalVM compiler, while the Native-Image deployment compiles Java code ahead of time into a native executable, skipping interpretation and compilation of Java code at run time.

In addition to JVM languages, GraalVM supports the execution of guest languages by defining interpreters written with the *Truffle framework* [21]. Truffle is an abstraction layer in GraalVM that provides a domain-specific language API based on Java annotations. It allows to define interpreters and uses *partial evaluation* [44, 15, 9, 28] to transform these interpreters into an intermediate representation that can be further optimized by the GraalVM compiler.

[1] https://github.com/smarr/are-we-fast-yet
[2] https://browserbench.org/JetStream/in-depth.html
[3] https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html

Partial evaluation must be performed at run time in both GraalVM deployments since it requires the input programs of guest language interpreters, which are not known ahead of time. Therefore, in the context of the Truffle framework, both GraalVM deployments support JIT compilation of partially-evaluated guest language interpreters. Consequently, in the Native-Image deployment of a guest language, the interpreter and runtime are AOT compiled, while the user code written in that guest language is still JIT compiled.

Through Truffle and partial evaluation, GraalVM provides official support for seven language runtimes with competitive performance [36, 37] for popular languages such as Python[4], JavaScript[5], C/C++ and others via LLVM bitcode[6] [33], WebAssembly[7], R[8], Ruby[9], and a meta-circular Java runtime called Espresso[10]. Apart from implementing different language specifications, these runtimes primarily differ in their internal data structures and interpreter implementations, including abstract-syntax-tree (AST) interpreters, bytecode interpreters, and hybrid approaches that combine aspects of AST and bytecode interpreters.

Our paper focuses on guest language runtimes using Truffle and partial evaluation in the Native-Image deployment of GraalVM. Therefore, the rest of this paper will refer to the Native-Image deployment when talking about GraalVM, the GraalVM compiler, or the Truffle language runtimes.

2.2 GraalVM Compiler

The *GraalVM compiler* [45] is a dynamic optimizing compiler used by GraalVM to compile multiple languages to highly optimized machine code. It contains numerous optimizations that apply platform-specific and platform-independent optimizations [13] and extensively uses *speculative optimizations* [11] based on *assumptions* [38]. If one of the assumptions no longer holds, the GraalVM compiler invalidates the generated machine code and transfers the execution back to the interpreter [38] through *deoptimization* [19].

In the context of guest language interpreters written with Truffle, GraalVM uses the GraalVM compiler as a just-in-time compiler to partially evaluate and compile guest language functions at run time [44]. For this purpose, the GraalVM compiler uses two different configurations called compiler tiers[11] [18]. Tier 1 focuses on compilation time and applies fewer short-running optimizations. Tier 2 focuses on optimal machine code and applies all optimizations available in the GraalVM compiler.

The GraalVM compiler's architecture consists of a *front end*, performing platform-independent compiler optimizations on a high-level intermediate representation (IR) called *GraalIR* [11, 10], and a *back end*, performing register allocation and code generation on a low-level IR called *LIR* [13, 41, 23]. The front end further consists of a *high tier*, *mid tier*, and *low tier*, performing optimizations on different abstraction levels. When compiling guest language functions, the *truffle tier*, performing partial evaluation, precedes the front end.

---

[4] https://github.com/oracle/graalpython
[5] https://github.com/oracle/graaljs
[6] https://github.com/oracle/graal/tree/master/sulong
[7] https://github.com/oracle/graal/tree/master/wasm
[8] https://github.com/oracle/fastr
[9] https://github.com/oracle/truffleruby
[10] https://github.com/oracle/graal/tree/master/espresso
[11] https://docs.oracle.com/javase/8/docs/technotes/guides/vm/performance-enhancements-7.html

Listing 1 Definition of a scope, timer, and counter with the help of a try-with-resources statement.

```
1  TimerKey timer = DebugContext.timer("Timer");
2  CounterKey counter = DebugContext.counter("Counter");
3
4  void run(Graph graph) {
5      DebugContext debug = graph.getDebugContext();
6      try (DebugContext.Scope scope = debug.scope("Phase");
7              DebugCloseable t = timer.start(debug)) {
8              ...
9              counter.increment(debug);
10             ...
11     }
12 }
```

2.3 GraalVM Compiler Debug Interface

The *GraalVM compiler debug interface*[12] in the GraalVM compiler enables logging, IR dumping, and the extraction of compilation-time metrics, such as the execution time of the compiler front end, the number of allocated bytes in the compiler mid tier, or the number of generated `mov` instructions during register allocation. The debug interface provides *timers*, for tracking execution time, *memory usage trackers*, for tracking memory usage (i.e., the number of allocated bytes), and *counters*, to keep track of arbitrary counts. This allows developers to get different metric values for parts of the compiler. Developers assign unique names to these metric values and define their measurement scopes with one or several *keys*. Keys allow combining measurements of several compiler regions into one metric value and are combined based on their name. For example, if a compiler performs parts of the same transformation in two different compiler locations and we want to measure the total time of this transformation, we can use two keys with the same name to combine both scopes into the same metric value. A *debug context* stores instances of each metric value. Debug contexts exist once per compilation, and the compiler passes them through all phases. This design enables a per-compilation extraction of metric values.

The debug interface provides *scopes* to enable the logical grouping of keys. Scopes have names and can be nested into one another. The debug context contains helper methods to open scopes while the closing is performed automatically with *try-with-resource* statements, as shown in line 6 of Listing 1. Scopes, timer keys, and memory usage trackers use this automatic closing mechanism, while counter keys need to be manually incremented by the developer, as shown in line 9.

Listing 1 provides an example for the use of the debug interface. When a developer creates a timer key with the help of the `DebugContext` class, as shown in line 1, the call to `timer()` allocates a new key object (`TimerKey`) and links it to a metric value based on the provided name. When the runtime starts the timer in line 7, the key object forwards the name of its metric value to the debug context, which initializes the timer value. After the runtime exits the `try` block in line 11, the timer stops and updates its value in the debug context. At the end of the compilation, the compiler extracts the metric values of all debug contexts and exports them to the console or a file.

[12] `https://github.com/oracle/graal/blob/master/compiler/docs/Debugging.md`

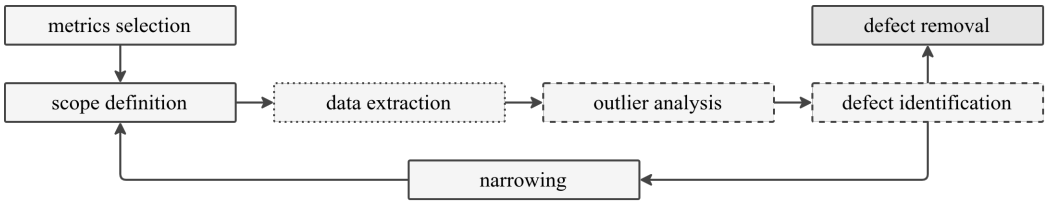


Figure 2 Abstract representation of ICON, consisting of three main phases represented by different border styles.

The GraalVM compiler provides a `Timers` option alongside counterparts for memory usage and counters to enable the extraction of specific metrics. Developers pass these options to the compiler at program startup, which enable metric values based on their name or the name of an enclosing scope. If a key is disabled, the compiler does not extract data for the associated metric value.

3 ICON: Iterative Compilation-Time Optimization Through Outlier-Driven Narrowing

We propose ICON, a new data-driven approach to compilation-time optimization, a methodology focusing on outliers to identify potential problems in compiler implementations. The key idea of ICON is to split up high-level metrics, such as total compilation time or overall memory footprint, into individual *extraction scopes*, such as compiler optimizations, compilations of individual functions, or even into individual instruction in the compiler source code. ICON tries to find defect locations by iteratively narrowing extraction scopes until developers can identify the source of a problem. An outlier analysis after every iteration drives the narrowing process and identifies which extraction scopes to refine. Figure 2 shows an abstract depiction of ICON.

ICON consists of three steps, including the *metric and benchmark selection step* (solid border), the *data extraction step* (dotted border), and the *outlier analysis and defect identification step* (dashed border). To explain the methodology process depicted in Figure 2, consider the hypothetical implementation of a compiler phase `p` in the compiler back end of an existing compiler. Figure 3 shows a fragment of the resulting compiler architecture.

For the performance evaluation of `p`, the initial step of ICON is to pick representative benchmarks and metrics, such as compilation time or allocated memory, and to select an initial set of extraction scopes for the first iteration. In our example, the entire compilation pipeline represents a good starting point, as shown by scope `s0` in Figure 3. After applying the compiler code changes and enabling the relevant timers and counters, the next step is to extract a data set based on the selected extraction scopes with the help of the benchmarks. An outlier analysis follows the data extraction to identify outlier compilations via statistical analyses. This outlier analysis results either in a concrete source code location responsible for the outliers or at least in a direction to narrow the selected extraction scopes.

If the outlier analysis points in a new direction for the next iteration, the next step is to select a new set of extraction scopes that narrow the existing ones and to repeat the extraction and analysis process. In our example, a meaningful narrowing is extracting data for the compiler back end, as shown by scope `s1` in Figure 3.

If the data set of the next iteration contains evidence for the previous outliers, the process continues by refining the extraction scopes. Otherwise, the process continues at the previous iteration by narrowing the existing extraction scopes differently. Consequently, since we

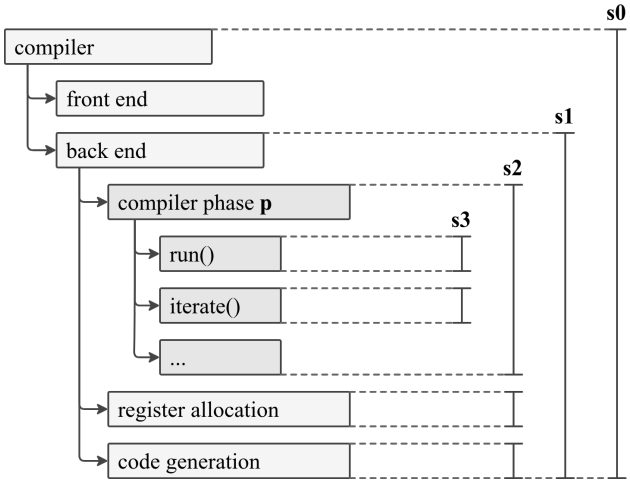


Figure 3 Fragment of a compiler pipeline structure after introducing a new compiler phase `p`.

assume that the narrowing in our example succeeded, the process continues with extraction scopes `s2` searching for outliers in individual compiler phases, including `p`, and scopes `s3`, searching for outliers in individual compiler functions in `p`.

Extraction scopes can theoretically be narrowed down to individual instructions in the compiler source code. The process ends when developers have a clear enough view about the defect locations responsible for the outliers or cannot find any outliers in the data.

While the outlier analysis is essential to ICON, we do not specify a concrete outlier-detection algorithm. Instead, methodology adopters can define concrete algorithms based on the requirements and properties of their compilers. We describe the approach we used for outlier detection in GraalVM in Section 4.4.

Although we focus on a manual inspection and outlier analysis in this paper, an automatic outlier detection and narrowing process could enhance our approach. This automation would aid in integrating our methodology into existing testing and benchmark infrastructures. However, defining such an automated process is outside the scope of our paper.

In the following, we present a detailed definition of *extraction scopes* and describe the necessary changes in GraalVM to support ICON.

3.1 Extraction Scopes

Extraction scopes define the granularity of the data extraction as tuples consisting of a *temporal* and a *spatial* component. The temporal component defines how to split or aggregate the entire execution time of the compiler into individual metric values. Examples are:

- *per execution*. This results in one metric value for the entire run time of the compiler. An example is the extraction of compilation time to compare against historical data.
- *per iteration*. This requires that the program under test executes several iterations and results in one metric value for every iteration. An example is the extraction of compilation time for every benchmark iteration to check their consistency over time.
- *per compilation unit*. This represents the usual approach to metric extraction and results in one metric value for each compilation unit. The metric values depend on the compiler's definition of compilation units (e.g., every function, every module, etc.).
- *per compiler tier*. This is a refinement of *per compilation unit* and results in one metric value per compiler tier. An example is the compilation-time extraction of a compiler phase for every compiler tier to compare the time spent in each compiler tier.

The spatial component defines how to split or aggregate the compilation pipeline (i.e., the compiler source code) into individual metric values. Examples are:

- *per pipeline*. This results in one metric value for the entire compiler pipeline. An example is the extraction of the maximum memory consumption of the compiler when running a benchmark to check that it does not exceed a predefined threshold.
- *per compiler phase*. This results in one metric value for each selected phase in the compiler pipeline. An example is the extraction of compilation time per compiler phase to find optimization potential.
- *per function*. This results in one metric value for each selected function in the compiler source code. An example is the memory-consumption extraction of all functions in a phase that allocates too much memory to find the exact allocation location.
- *per instruction*. This results in one metric value for each selected source code instruction. An example is the extraction of memory allocation for selected source code lines in the compiler.

The temporal and spatial components depend on the compiler architecture and might be linked based on the compiler's design.

3.2 Implementation in GraalVM

GraalVM was already well fitted to support our methodology. However, in terms of data extraction, some critical parts were missing to support a fine-grained metric value extraction.

The initial implementation of the GraalVM compiler debug interface, as explained in Section 2.3, focuses primarily on the extraction of metric values on a per-name basis. Metrics are extracted for every compiler phase in the GraalVM compiler, so the metric values are associated with the names of the compiler phases. This represents a limitation, since multiple executions of the same compiler phase, which is common in the GraalVM compiler, are merged into a single metric value. An example is the *canonicalizer phase*, which transforms guest language constructs into a canonical form and is executed many times throughout a compilation. Even though these phase executions should be treated independently, the initial debug interface implementation merges their metric values.

To remove this limitation, we extended the existing concept of scopes in the GraalVM compiler debug interface to *aggregate*, *unique*, and *singleton* scopes. Aggregate scopes are similar to the existing scope implementation. As their name suggests, they combine the values produced by several scope executions into a single metric value. Unique scopes on the other hand separate the values of individual scope executions into separate metric values by assigning them a unique name. Singleton scopes ignore the nesting of previous scopes and produce a single metric value, regardless of the scopes they are nested in.

Consider the example in Listing 2. Since the timer key `a_t` in line 5 is part of an aggregate scope, all its executions will contribute to the same metric value *A.Timer*, even though the loop is executed three times. The code in line 12 however, will result in three individual metric values *U_1.Timer*, *U_2.Timer*, and *U_3.Timer*, since the timer key `u_t` is inside a unique scope. Although timer keys `s_t` in lines 7 and 14 are nested in the different outer scopes *A* and *U*, all their executions contribute to the same metric value *S.Timer*, since their nearest nesting scope is a singleton scope.

As a result of using these new scope types, it is possible to extract metric values independently for multiple executions of the same compiler phase. This allows a more precise analysis of the time spent in individual compiler phases and supports the fine-grained metric value extraction required for ICON.

Listing 2 Definition of aggregate, unique, and singleton scopes inside a loop.

```
void run (Graph graph) {
  DebugContext debug = graph.getDebugContext();
  for (int i = 0; i < 3; i++) {
    try (DebugContext.Scope a = debug.aggregateScope("A");
          TimerKey a_t = a.timer("Timer").start(debug)) {
      try (DebugContext.Scope s = debug.singletonScope("S");
            TimerKey s_t = s.timer("Timer").start(debug)) {
        ...
      }
    }
    try (DebugContext.Scope u = debug.uniqueScope("U");
          TimerKey u_t = u.timer("Timer").start(debug)) {
      try (DebugContext.Scope s = debug.singletonScope("S");
            TimerKey s_t = s.timer("Timer").start(debug)) {
        ...
      }
    }
    ...
```

4 GraalVM Compilation-Time Evaluation

Based on ICON introduced in Section 3, we extracted compiler metrics for the GraalVM compiler. While our approach primarily focuses on finding defects in compiler implementations, its implementation in GraalVM also enables a detailed view of the distribution of compiler metrics across different compiler phases in the GraalVM compiler. We extracted compilation time and memory usage for all compiler phases based on a large set of benchmarks. In the context of this paper, we will focus on the evaluation of compilation time. The extracted data contains metric values for five partial-evaluation-based language runtimes on GraalVM and allows us to present the first detailed look into the distribution of compilation time across compilation phases in GraalVM. This distribution represents a good starting point for our evaluation and drives the narrowing required for the outlier analysis in Section 4.5.

We start this section by introducing the set of language runtimes we chose to get a representative data set of compilation-time metrics, then proceed by describing the used benchmarks and setup. We conclude by explaining the data extraction process, presenting our results, and performing an outlier analysis on the data.

4.1 Languages

We chose a set of five language runtimes based on the Truffle framework to get a representative data set of the compilation-time distribution in partial-evaluation-based languages compiled by the GraalVM compiler. The runtimes differ in their interpreter implementation and the type system of their implemented languages. Both impact the compilation-time distribution in the GraalVM compiler. Based on the available interpreter implementations and type systems, we chose GraalJS[13], GraalPy[14], Espresso[15], GraalWasm[16], and the GraalVM LLVM Runtime[17], as further described in Table 1.

[13] https://github.com/oracle/graaljs
[14] https://github.com/oracle/graalpython
[15] https://github.com/oracle/graal/tree/master/espresso
[16] https://github.com/oracle/graal/tree/master/wasm
[17] https://github.com/oracle/graal/tree/sulong [33]

Table 1 GraalVM language runtimes used for the extraction of compilation-time data.

Runtime	Language	Type system	Interpreter
GraalJS	JavaScript	dynamic, weak typing	AST interpreter
GraalPy	Python	dynamic, strong typing	bytecode interpreter
Espresso	Java	static, strong typing	bytecode interpreter
GraalWasm	WebAssembly	static, strong typing	bytecode interpreter
GraalVM LLVM Runtime	LLVM bitcode	static, strong typing	hybrid interpreter (combines aspects of AST and bytecode interpreters)

4.2 Benchmarks and Setup

We used a set of 94 benchmarks from the "Are We Fast Yet?"[18] [27], JetStream 2[19], "Computer Language Benchmark Game,"[20] and several internal benchmark suites for our evaluation. The benchmarks represent real-world computing tasks and are already used internally by different GraalVM language teams to evaluate peak performance, interpreter speed, and memory usage of their language runtimes.

We used "Are We Fast Yet?" for evaluating Espresso, the Java runtime implemented in Truffle. It contains 14 benchmarks focusing on several computing areas, such as JSON string parsing, the computation of the Mandelbrot set, and physics simulations [27].

We used JetStream 2 for evaluating GraalJS and executed 15 of the 64 available benchmarks. These focus on cryptography, physics simulations, or PDF processing. Most excluded benchmarks require browser-specific functionality unavailable in standalone JavaScript engines or WebAssembly support. Since we measured our WebAssembly runtime separately, we did not want to pollute GraalJS results with WebAssembly compilations.

We used the "Computer Language Benchmark Game" [27] benchmarks in combination with other open source benchmarks (*bzip2*[21], *gzip*[22], *stockfish*[23], *oggenc*[24]) to evaluate GraalPy, the Python runtime, and the GraalVM LLVM runtime. Based on the language runtime, we selected a subset of the available benchmarks used by the respective GraalVM language teams[25]. Consequently, we used 34 benchmarks for evaluating GraalPy, and 18 for the GraalVM LLVM runtime. The benchmarks include compression algorithms, chess simulations, physics simulations, and scheduling tasks. We compiled the GraalVM-LLVM-runtime benchmarks with a custom LLVM toolchain[26] based on the *Clang*[27] compiler 16.0.1.

For evaluating GraalWasm, we used an internal benchmark suite consisting of 13 benchmarks. These include the Digitron benchmark, an AST interpreter for arithmetic expressions, the FFT [4] benchmark, computing the Fast Fourier transform, and the Phong [31]

[18] https://github.com/smarr/are-we-fast-yet
[19] https://browserbench.org/JetStream/in-depth.html
[20] https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html [27]
[21] https://sourceware.org/bzip2/
[22] https://www.gzip.org/
[23] https://github.com/official-stockfish/Stockfish
[24] https://xiph.org/ogg/
[25] https://github.com/oracle/graalpython/tree/master/graalpython/com.oracle.graal.python.benchmarks
[26] https://github.com/oracle/graal/blob/master/sulong/docs/contributor/TOOLCHAIN.md
[27] https://clang.llvm.org/

benchmark, computing a shading model for a 3D scene. All of them are written in C and compiled with the *WASI SDK*[28] version 21.0 based on the *Clang* compiler version 17.0.0. The benchmark source code is publicly available as part of the GraalWasm repository[29].

We executed all benchmarks on the Community Edition (CE) and the Enterprise Edition (EE) of GraalVM. We conducted the execution on an Intel Core i7-8750H with six cores at a fixed CPU frequency of 2.30 GHz and turbo boost disabled. The system has 32GB of main memory and runs Fedora Linux 38 (Workstation Edition).

We compiled all runtimes with *LabsJDK CE 21*[30] (`labsjdk-ce-21-jvmci-23.1-b22`). The GraalVM compiler performs individual compilations in separate threads with a separate memory space. We used sufficient iterations to ensure that the GraalVM compiler compiled all relevant functions in each benchmark and ran each benchmark 6 times in separate processes. We used geometric means across all 6 runs to account for measurement inaccuracies due to garbage collection and non-deterministic optimization phases. The resulting numbers represent the evaluation of the Native-Image deployment of GraalVM CE and EE.

4.3 Data Extraction

We surrounded the `run` method, the entry point of each compiler phase, of all compiler phases in the GraalVM compiler with a unique scope, as introduced in Section 3.2, to extract compilation-time data. This implementation allows us to get an individual metric value per phase execution. We put *method inlining* and the *graph decoding* performed during partial evaluation into aggregate scopes since we are not interested in inlining and decoding metrics of individual functions but in their overall impact.

For our evaluation, we extracted the compilation time of every extraction scope in microseconds and computed the compilation time relative to the total compilation time. This normalization is necessary to account for the skewness of the raw data caused by the overrepresentation of short-running compilations in the data set. We extracted metric values for all compilations in our benchmarks and used the compilations of those source program functions that appeared in at least 3 of the 6 benchmark runs to compute the total compilation time for our evaluation. This preselection is necessary to avoid polluting the data with one-time compilations that might represent outliers due to garbage collection pauses or other non-deterministic influences happening exactly during this one compilation.

4.4 Evaluation

We start our evaluation with a high-level overview of the compilation-time distribution in GraalVM. The selected extraction scopes reflect the GraalVM compiler architecture and consist of *partial evaluation*, *compilation*, and *code installation*. We further divide the compilation into a *front end* and *back end*, and the front end into *high tier*, *mid tier*, and *low tier*. We chose the selected granularity to keep the resulting plots clear and readable. In the context of our plots, we define *total compilation time* as the sum of partial evaluation, compilation, and code installation.

[28] `https://github.com/WebAssembly/wasi-sdk`
[29] `https://github.com/oracle/graal/tree/master/wasm/src/org.graalvm.wasm.benchcases/src/bench`
[30] `https://github.com/graalvm/labs-openjdk-21`

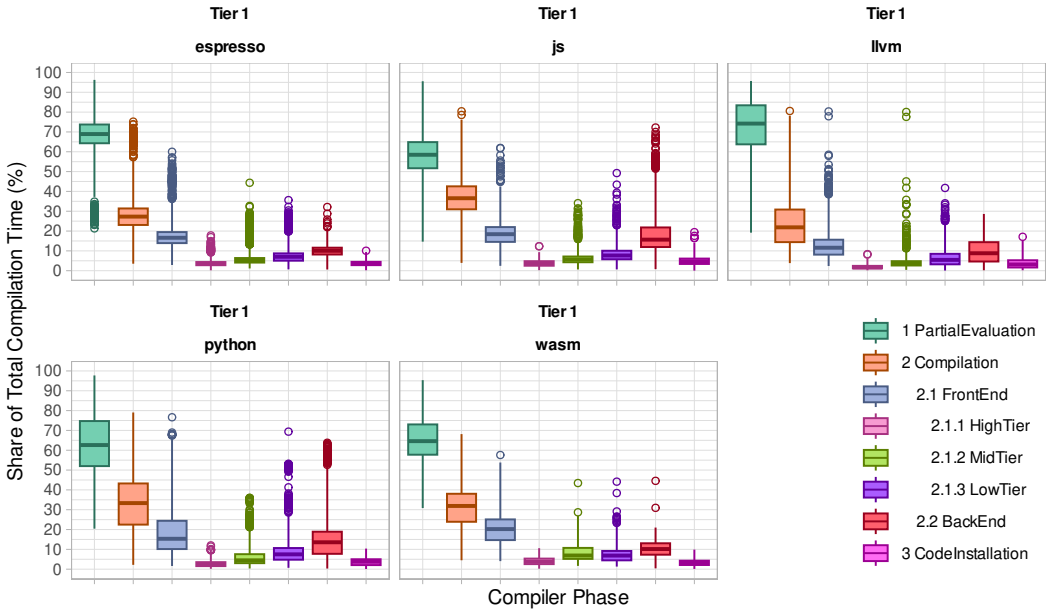


Figure 4 Compilation-time distribution of Tier-1 compilations in GraalVM CE and EE. Compiler phases are from left to right based on their index.

We show the results of the first extraction step in Figure 4, showing the compilation-time distribution of Tier-1 compilations in GraalVM CE and EE. Since their configuration is identical in Tier 1, we combined both editions into a single plot. Figure 5 shows GraalVM-CE Tier-2 compilations, and Figure 6 shows GraalVM-EE Tier 2. The plots show the selected extraction scopes from left to right based on their execution point in the compilations.

Figure 4 shows that partial evaluation has the highest impact on the total compilation time in Tier-1 compilations, with a median between 58.5% and 72.7%. It is also apparent that the median front-end time, in the range of 11.60% to 20.20%, is higher than the back-end time, 8.81% to 15.70%, while the average low-tier time, 5.46% to 7.49%, is higher than the high-tier, 1.66% to 3.79%, and mid-tier times, 3.41% to 6.90%.

Figure 5 shows that the difference in average time between front end and back end is larger in Tier-2 compilations, 7.88%pt (percentage points) to 15.10%pt, than in Tier 1, 1.70%pt to 10.05%pt. This observation seems reasonable since the back-end compiler phases are nearly identical in Tier-1 and Tier-2 compilations, whereas the GraalVM compiler applies additional, longer-running optimization phases in the front end of Tier 2. We verified this conclusion based on absolute numbers to make sure the back-end time stays consistent while the front-end time increases. The only exception is WebAssembly, were the difference between front end and back end is larger in Tier 1 (10.05%pt) than in Tier-2 CE (7.83%pt).

The difference between Tier-1 and Tier-2 front-end and back-end times becomes even more apparent in Figure 6, since Tier-2 EE, with differences in the range of 22.36%pt to 29.11%pt, has additional optimization phases compared to Tier-2 CE. Furthermore, compilation takes an equal amount of time or longer than partial evaluation in Python, with 48.71% and 49.19%, and JavaScript, 50.27% and 46.33%. This can again be explained by the increase in font-end time.

We try to identify possible outliers that require further investigation based on the extracted data. We define *outliers* based on the *interquartile range method* [16] and focus on *extreme* outliers that are more than 3 interquartile ranges (IQR) away from the first (Q1) and third quantile (Q3). We represent outliers in the plots by circles above and below the boxplots.

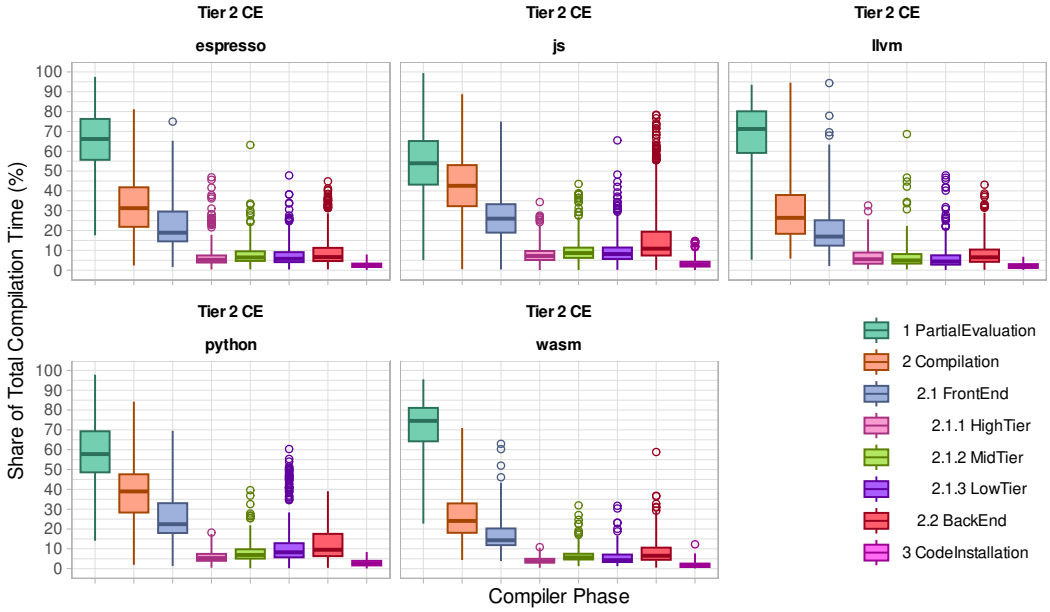


Figure 5 Compilation-time distribution of Tier-2 compilations in GraalVM CE. Compiler phases are from left to right based on their index.

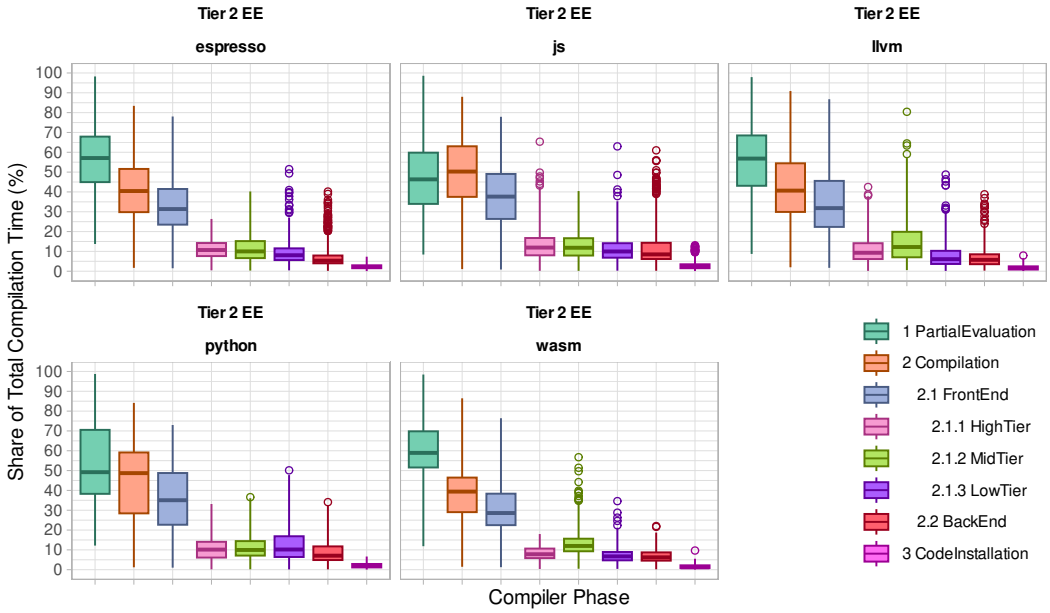


Figure 6 Compilation-time distribution of Tier-2 compilations in GraalVM EE. Compiler phases are from left to right based on their index.

All three data sets contain many outliers within individual extraction scopes. For example, Espresso and JavaScript contain a lot of outliers in all extraction scopes of Tier-1 compilations, while the GraalVM LLVM runtime shows significant outliers in the mid tier. In Tier-2 compilations, JavaScript presents significant outliers in the back end, while Python contains several outliers in the low tier of CE compilations. The GraalVM-LLVM-runtime outliers found in the mid tier of Tier-1 compilations are also found in Tier 2.

Analyzing all outliers in all data sets would be possible. However, we will focus on the most significant outliers to identify defects and optimizations with a high impact potential. To select the relevant outliers, we prioritize them based on their distance from $Q1$ or $Q3$ in terms of multiples of the IQR and look into extraction scopes with a high outlier density.

Consequently, we will analyze the outliers in the mid tier of GraalVM-LLVM-runtime compilations. Especially the outliers in Tier-1 compilations have a distance of more than $30IQR$ from $Q3$. The outliers are consistent throughout all editions and compiler tiers and are responsible for up to 80% of the total compilation time in Tier 1 and Tier-2 EE.

Furthermore, we will analyze the back end of JavaScript compilations. This extraction scope shows a high outlier density throughout all editions and compiler tiers. The same applies to the low tier of Python Tier 1 and Tier-2 CE compilations.

Although it is hard to quantify partial evaluation as an outlier, it is apparent from a visual analysis of the plots in Figures 4 to 6 that partial evaluation takes up a significant part of the total compilation time in all compiler editions and tiers. Therefore, in addition to our outlier analyses, we show the applicability of our methodology for finding optimizations in existing compiler phases by analyzing partial evaluation in all guest language runtimes.

4.5 Outlier Analysis

We perform an outlier analysis based on the possible defects identified during the evaluation of the GraalVM compilation time in Section 4.4. We verify the defects and identify their source code locations. In addition, we show that our approach can find optimization locations in compilers by focusing on long-running extraction scopes.

4.5.1 Mid Tier in the GraalVM LLVM Runtime

We first analyzed the outliers in the GraalVM-LLVM-runtime mid tier and identified the benchmark functions responsible for the outliers to find a possible defect in the compilations. The outliers were two functions in the *bzip2* and *oggenc* benchmarks that did not show any suspicious characteristics in the C source code. Next, we refined our extraction scopes from a top-level view of the mid tier to individual compiler phases. Figure 7 shows the result of the narrowed extraction scopes of Tier-1 compilations. The figure shows that the *frame-state-assignment phase* has a lot of outliers and the previously identified functions indeed spend nearly all of their mid-tier time in the frame-state-assignment phase of Tier-1 compilations. We also verified that the frame-state-assignment phase is the defect source in Tier-2 compilations (omitted from the paper).

Frame states are a mapping from machine state (i.e., registers and native stack frames) to interpreter state (i.e., JVM stack frames) and are required by GraalVM during deoptimizations to regenerate the interpreter state of a program [11, 12]. During partial evaluation, the GraalVM compiler generates a *frame-state* node for every operation that changes the local state of a method (local variables, operand stack values, and locked objects) and attaches it to the node in the GraalIR graph that causes this change. Subsequent compiler optimization phases might also introduce new nodes, and therefore new frame states. When entering

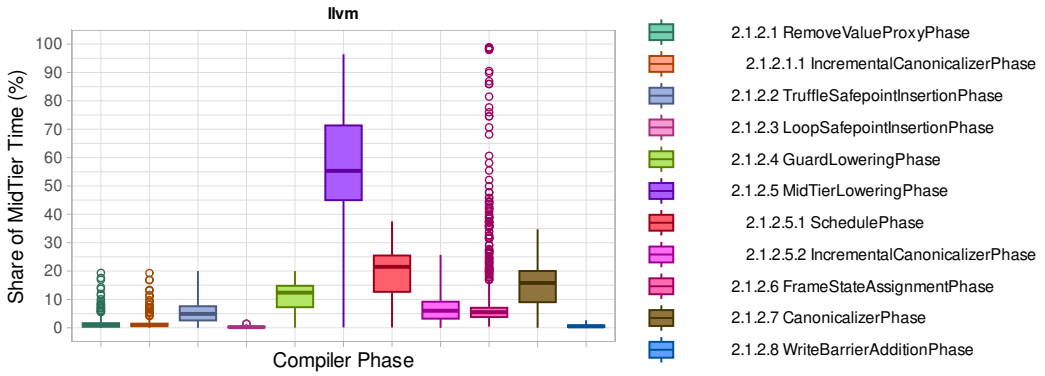


Figure 7 Compilation-time distribution of the mid tier in Tier-1 compilations. Compiler phases are from left to right based on their index.

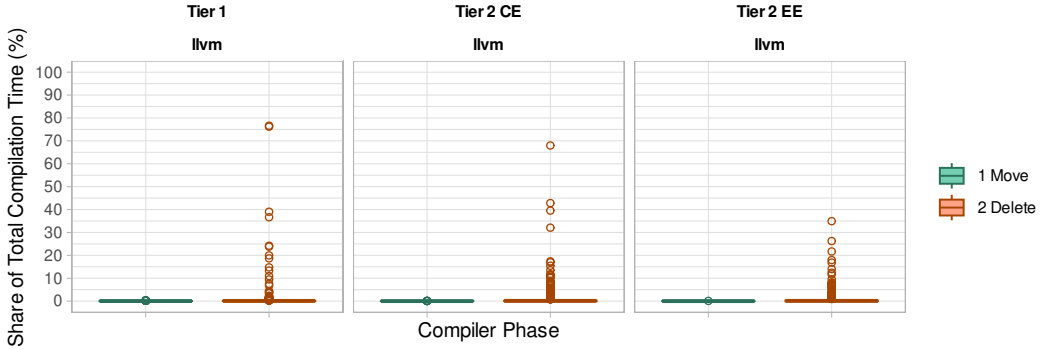


Figure 8 Compilation-time distribution of the two frame-state-assignment phase parts Move and Delete. Compiler phases are from left to right based on their index.

the frame-state-assignment phase, the graph is stable and no more nodes that may cause deoptimizations can be introduced in later compiler phases. Since the frame states are only required by deoptimizations, the frame-state-assignment phase moves the frame states from their originating nodes to deoptimization nodes at deoptimization points in the graph and removes unused frame states.

To further narrow the defect location, we introduced individual timers for the two source code method calls (*Move* and *Delete*) in the `run` method of the frame-state-assignment phase. The *Move* method moves existing frame-state nodes, while the *Delete* method deletes unused frame-state nodes. Figure 8 shows the result of this last narrowing step. The values of these last extraction scopes are relative to the total compilation time. The figure shows that the deletion of unused frame-state nodes is responsible for the outliers and should be optimized, as described in Section 5.1.

4.5.2 Back End in JavaScript

We analyzed the outliers in the JavaScript back end and identified that all functions responsible for the outliers are part of the *typescript* benchmark, which compiles a large TypeScript application to JavaScript. The benchmark uses the identified functions to walk the AST of the TypeScript application. The functions themselves are small and call each other recursively. As a result of these recursive calls, the GraalVM compiler can perform a lot of inlining.

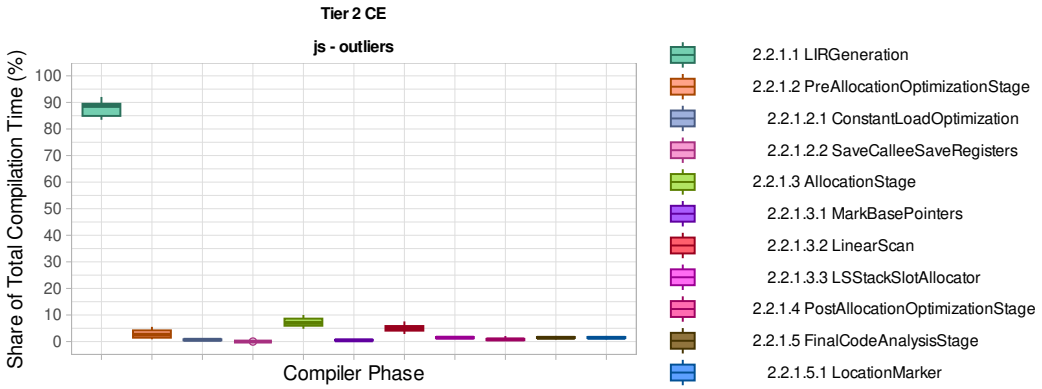


Figure 9 Compilation-time distribution of the back end in Tier-2 CE compilations. Compiler phases are from left to right based on their index.

We narrowed the extraction scopes to individual compiler phases in the back end and found that the identified functions on average spend 87% of their Tier-2 CE back-end time in *LIR generation*, as shown in Figure 9. Tier-1 (60%) and Tier-2 EE (82%) compilations produced similar numbers. The LIR-generation phase transforms high-level GraalIR nodes into low-level LIR nodes by iterating over the GraalIR graph and transforming each node.

We found out that the outlier functions spend most of their time transforming nodes that generate a *LIR frame state* during LIR generation. The LIR frame states represent garbage collection and deoptimization information and the GraalVM compiler computes them based on the frame-state nodes in the GraalIR graph by iterating over all values in the frame state and all its parent frame states and transforming the GraalIR representation of the frame-state values into LIR representations. A frame state has a parent frame state if its method was inlined into another method. The parent frame state represents the frame state of the method into which this method was inlined. We confirmed that the outlier functions had a lot of frame states as a result of inlining and, through a last narrowing step, confirmed that the identified outliers spend most of their time generating LIR frame states.

Optimizing this pattern would require significant changes in the compiler architecture which was not in the scope of this paper. We reported our findings to the GraalVM compiler team for further investigation.

4.5.3 Low Tier in Python

We analyzed the outliers in the Python low tier and found out that all outliers are instances of the same function in all benchmarks, the *time* function in the Python *time* module[31]. This function is built into GraalPy and returns the current time in seconds. The benchmarks use this function to measure execution time.

We narrowed the extraction scopes to individual compiler phases in the low tier and found that the identified functions spend more than 95% of the low-tier time in the *low-tier-lowering phase*. Lowering transforms nodes on a higher abstraction level to node structures on a lower abstraction level. For example, a *load-field* node, accessing a field on an object, is lowered to a *read* node, reading a value from an address in memory.

[31] `https://docs.python.org/3/library/time.html#time.time`

We also extracted the node count for the graph of the *time* function and found that it is a small graph with only 157 nodes. We narrowed the extraction scopes to the lowering of individual node types and found that four individual nodes were responsible for the time spent in lowering. These were two *exception-object* nodes, one *truffle-safepoint* node, and one *new-instance* node. The commonality between these nodes is that they are all lowered with the help of *snippets* [35]. The GraalVM compiler creates these snippets the first time they are used and caches them based on their compilation unit. This implies an initial overhead that is, in most cases, amortized by several usages of the cached snippets. However, since the graphs for the *time* functions only contain one or two of the snippet-lowered nodes, this results in a significant run-time overhead during the low-tier-lowering phase.

Optimizing this process would again require significant changes to the implementation of snippets and the lowering process itself, which is out of the scope of this paper. We reported our findings to the GraalVM compiler and GraalPy teams for further investigation.

4.5.4 Partial Evaluation

The partial-evaluation time of compilations is a long-standing problem in the GraalVM compiler and, as our evaluation shows, is responsible for up to 73% of the total compilation time. Therefore, finding an optimization opportunity in partial evaluation would improve a large portion of the overall compilation time in GraalVM guest language interpreters.

We narrowed the extraction scopes to sub-phases of partial evaluation to find possible optimizations to partial evaluation. The sub-phases include *decode-graph*, *cleanup-graph*, two *post-partial-evaluation suites*, *decode-inline-graph*, and the *inline post-partial-evaluation suite*, apart from several other optimization phases and sub-phases. Figure 10 shows the result of the narrowed extracted scopes. The plots only show the sub-phases with the highest compilation-time impact to improve readability.

Figure 10 shows that *graph decoding* (*DecodeGraph*, *DecodeInlinedGraph*) has the highest impact on compilation time spent in partial evaluation. Graph decoding iterates over a graph representation of the program IR, processing one node after the other. During the decoding, the GraalVM compiler applies several optimizations based on the type of each node. *Load-field* nodes for example, can be constant folded, while *invoke-with-exception* nodes, representing function calls, can be inlined. Partial evaluation performs graph decoding on the main partial-evaluation function (*DecodeGraph*), the function for which Truffle requests partial evaluation, and all functions inlined during partial evaluation (*DecodeInlinedGraph*). In Tier-2 compilations, graph decoding of inlined functions dominates the partial-evaluation time due to aggressive inlining policies, while in Tier 1, inlining is restricted.

We extracted the time spent processing specific node types during graph decoding by adding singleton scopes for all node types, as described in Section 3.2. Examples of node types are *load-field* nodes that load the field of an object, *if* nodes that represent an if statement, and *loop-begin* nodes representing a loop header. Figure 11 shows the result of the individual node types. The plots only show the node types with the highest impact on compilation time to improve readability.

Figure 11 shows that invoke-with-exception nodes dominate the graph-decoding process. Since invoke-with-exception nodes perform function inlining, it is expected that these nodes take up most of the total graph-decoding time. During inlining, invoke-with-exception nodes have to recursively decode the callee function and attach the resulting graph to the graph that is currently decoded. This process is already heavily optimized and most of its time is spent in the recursive decoding. Therefore, we did not further consider invoke-with-exception nodes for finding optimization potential. We instead focused on load-field nodes. We optimized the constant-folding performed for load-field nodes to improve the graph-decoding performance, as described in Section 5.2.

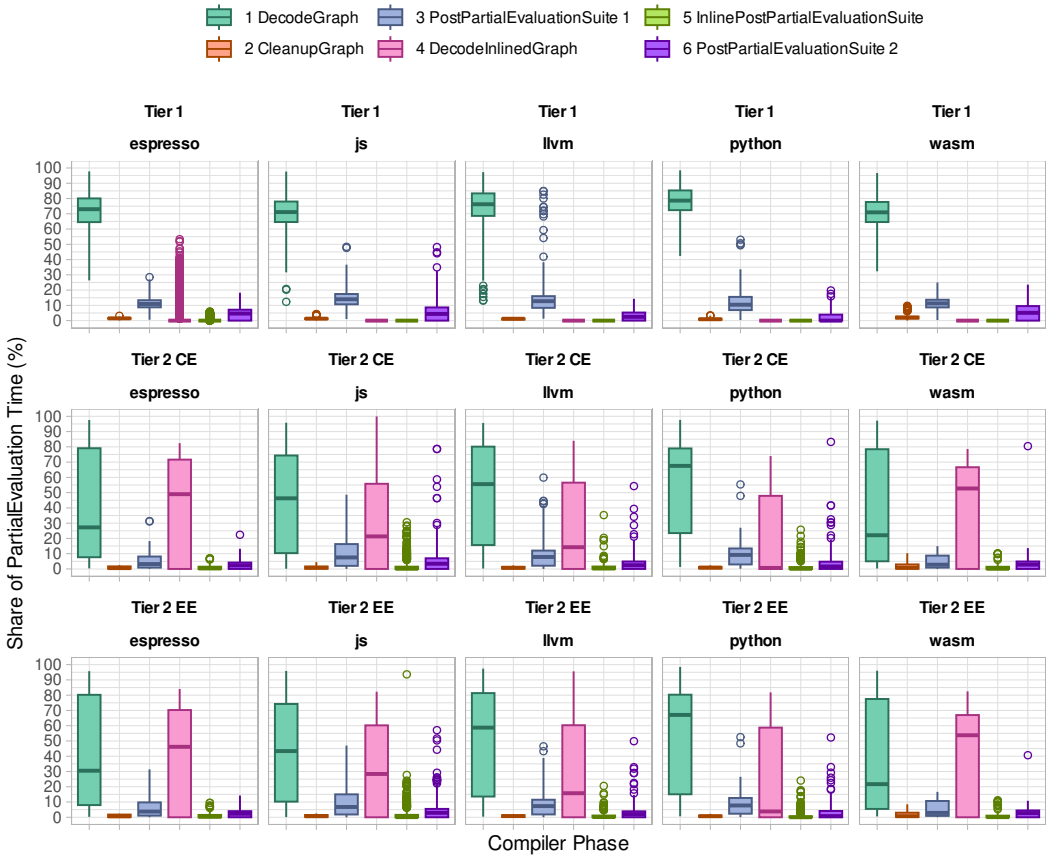


Figure 10 Compilation-time distribution of partial evaluation. Compiler phases are from left to right based on their index.

5 Optimization

5.1 Frame State Assignment – Optimizing Deletion Strategies

Graphs are a common way of implementing the intermediate representation of a compiler [39, 8, 7]. The design of these graphs is crucial because optimizations require an efficient way of traversing and manipulating the IR. The efficiency of these operations depends on the data structures used to represent nodes and edges in the graph.

GraalVM uses a directed graph that represents data flow and control flow in a single data structure [11, 12]. To model data flow, a node has *input edges*, pointing from the node using a value to the node defining this value. The reverse edges, so-called *usage edges*, which point in the opposite direction, are automatically maintained by the graph, so optimizations do not have to deal with maintaining them. However, in contrast to input edges, usage edges are not ordered and can only be accessed as an unordered set [11, 12]. Consequently, finding a specific usage of a node may require traversing the entire set. Unordered sets are a common way of reducing the memory footprint of the reverse-edge sets in JIT compilers[32].

[32] `https://chromium.googlesource.com/v8/v8/+/refs/heads/main/src/compiler/node.h#181`, `https://github.com/openjdk/jdk/blob/master/src/hotspot/share/opto/node.hpp#L319`

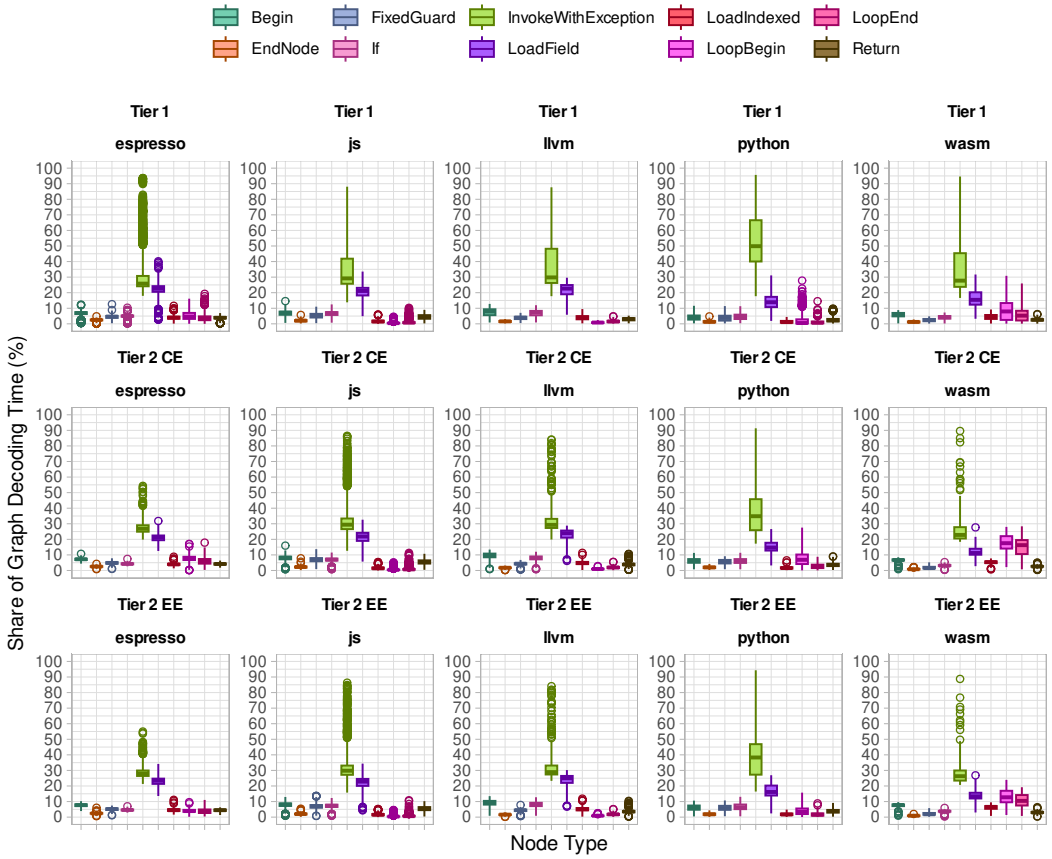


Figure 11 Compilation-time distribution of individual node types during graph decoding. Node types are from left to right in alphabetical order.

Usage edges are needed, for example, for deleting unused nodes in dead code elimination [29]. If a node is no longer used, its usage edges must be removed from all its inputs before the node can be deleted. As a result of deleting a usage of an input node, the input node itself can become unused, i.e., its usage set may become empty. This can lead to the transitive deletion of other nodes, which, in the worst case, requires several full traversals of potentially countless node usage sets.

For the frame-state-assignment phase of the GraalVM compiler, the outlier analysis in Section 4.5.1 showed that the deletion of unused frame-state nodes is responsible for a substantial part of the mid-tier time.

Normally, the traversal of usage sets during the deletion of frame-state nodes does not represent a problem, since the number of frame-state nodes is usually small. However, the graphs in which we identified outliers contained hundreds of frame-state nodes due to excessive inlining, resulting in slowdowns in the frame-state-assignment phase. Therefore, we updated the algorithm for deleting nodes. In the original implementation, every unused node was visited one after the other, and the usage sets of the node's inputs were updated immediately when deleting the node. This led to a lot of usage set traversals due to the transitive deletion of nodes.

Instead of updating the usage sets of nodes immediately, the updated algorithm tracks the usage counts of nodes in a separate map. The algorithm performs a depth-first traversal of the graph starting at the inputs of the deleted frame-state nodes, updates the usage counts

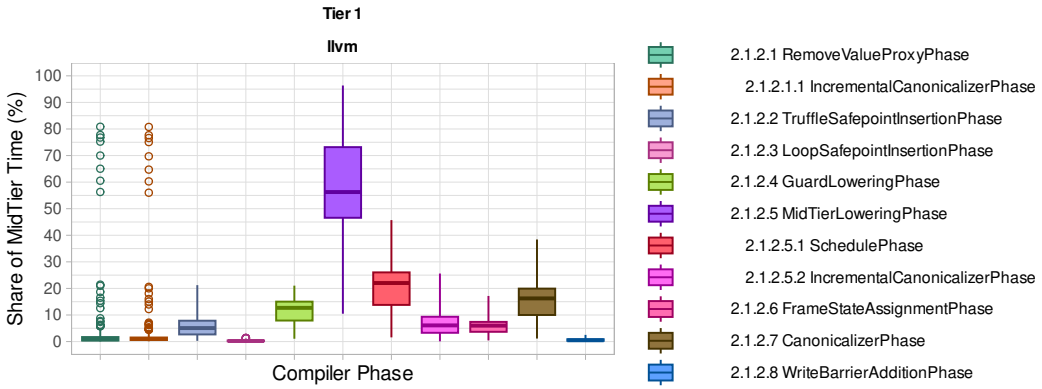


Figure 12 Compilation-time distribution of the mid tier in Tier-1 compilations after optimizing the frame-state-assignment phase. Compiler phases are from left to right based on their index.

in the map, and deletes unused nodes along the way. The traversal stops at nodes whose usage count in the map is non-zero. After the graph traversal, only nodes that are alive and have changed usage counts need to be updated. The pseudocode for the updated algorithm can be found in the appendix (Listing 3).

As a result of this optimized algorithm, the previous outliers no longer exist. We show the updated mid-tier compilation time of Tier-1 compilations in Figure 12. We also verified that the outliers no longer exist in Tier 2 (omitted from the paper). Due to the reduction of the time spent in the frame-state-assignment phase, new outliers in the *incremental canonicalizer phase* (2.1.2.1.1) arose that would be worth investigating, but are outside the scope of this paper. We confirmed that the new outliers were not caused by changes in the frame-state-assignment phase but were already present in the previous data set.

5.2 Graph Decoding – Constant-Fold Caches

Constant-folding is a crucial operation in compilers based on partial evaluation, because it allows these compilers to evaluate parts of the program at compilation time, and thus simplify the program [29, 1, 40]. When a partial-evaluation-based compiler identifies a constant in the program representation, many subsequent operations depending on that constant can also be replaced with constants. For example, a read operation from a field of a constant object (i.e., an object represented by a constant pointer), can be evaluated during compilation, and can be replaced with another constant that holds the value of the respective field.

Object-field reads and array reads on constant values are very common operations in interpreters, because the program representation is encoded in either ASTs or bytecode arrays [25], and the interpreter is partially evaluated for a given section of the program that is a constant value from the perspective of the compilation. Thus it is critical that constant-folding is executed very efficiently.

Reading a field from a constant object requires the compiler to (1) determine whether the field or an array location is guaranteed to remain constant after partial evaluation, and (2) to read the value from the respective offset in the program's memory. The first step usually relies on modifiers or annotations found in the source code of the interpreter. Both of these steps involve calls to the runtime environment, and rely on reflective metadata, which is usually expensive to obtain compared to a simple object-field read.

In the GraalVM compiler, the outlier analysis in Section 4.5.4 showed that the constant-folding of load-field nodes (which represent object-field reads in GraalIR) is responsible for a substantial part of the partial-evaluation time.

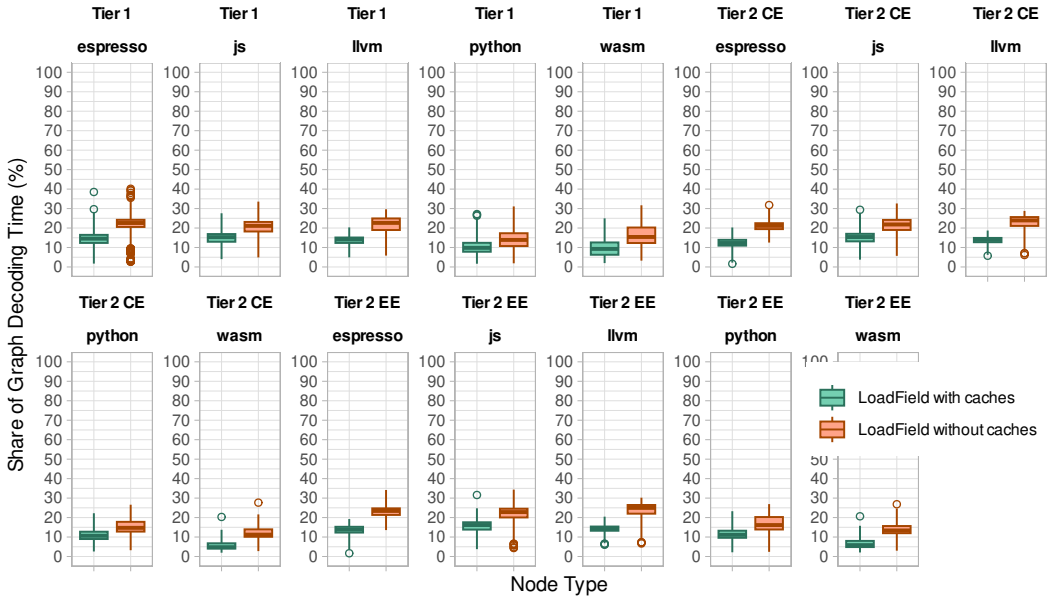


Figure 13 Comparison of time spent in load-field nodes during graph decoding before and after the introduction of caches. The time with caches is on the left in each plot, the time without caches on the right.

If the same constant occurs multiple times in an IR graph, it is represented by a single IR node to reduce memory footprint and compilation time [7, 8, 11, 12]. To maintain this graph property, the GraalVM compiler, before adding a new constant node to the graph, performs a graph traversal to look for equivalent nodes. This can be an expensive operation depending on the graph size.

In the current partial-evaluation implementation in the GraalVM compiler, all constant-folding attempts are independent. This means that the reflective metadata and value are loaded again and again regardless of whether this information was already retrieved by a previous constant-folding attempt. In addition, every constant-folding attempt allocates a new constant node that is discarded when the GraalVM compiler identifies an equivalent node in the graph. On average, the same constant field is read 8.5 times per compilation unit across all compiler tiers and editions. Therefore, to reduce the number of calls to the runtime environment as well as the number of allocated constant nodes and the number of graph traversals, we introduced a two-layer cache system in the constant-folding performed during partial evaluation.

As a result, the impact of constant-folding on partial evaluation was reduced, as shown in Figure 13. Overall, this optimization led to a compilation-time reduction between 2.25% (Python) and 6.88% (LLVM Runtime) in Tier 1, between 2.48% (Python) and 8.16% (WebAssembly) in Tier-2 CE, and between 4.49% (JavaScript) and 9.45% (Espresso) in Tier-2 EE.

6 Related Work

There is extensive research in the field of compilation-time optimization [22, 20, 30, 24, 2]. However, ICON, focusing on identifying outliers to find optimization opportunities in existing compilers, combines multiple aspects that we are not aware of being found together in any

sole research. It combines (1) iterative narrowing of scopes to analyze a problem with (2) focusing on outliers in extracted data to (3) improve compilation-time metrics. We, therefore, focus on related work in compiler optimization similar to ICON in at least one aspect.

Brown et al. [5], propose a data-driven methodology to identify the impact of compiler optimizations on security-oriented aspects of the generated machine code. They evaluate 20 benchmarks and analyze the availability of gadget sets used for code reuse attacks based on the enabled compiler optimizations in the GCC and Clang compilers. They perform a coarse-grained analysis based on optimization levels available in GCC and Clang, and a fine-grained analysis on individual optimizations in those compilers, similar to the narrowing of scopes in ICON. Furthermore, they use an outlier analysis to identify relevant compiler optimizations, similar to the outlier analysis defined in ICON.

Bryksin et al. [6], propose a method to identify code anomalies in order to find issues in compilers. They focus on source code fragments that are not typically found in a given programming language or uncharacteristic bytecode produced by a compiler. With anomaly detection algorithms, similar to the outlier detection in ICON, they identified several optimization opportunities in the Kotlin compiler.

Regarding compilation-time optimization, existing work can be categorized into approaches for phase selection and ordering [22, 30, 24, 2], automatic compiler optimization level selection [20], and automatic compiler heuristics or optimization tuning [14, 32]. Most existing approaches use machine learning and primarily focus on the common case instead of outliers.

6.1 ICON-like Approaches in Other Compilers

Based on online reports[33], compilation time is an important factor for many state-of-the-art compiler implementations. To improve these metrics, some compiler teams are actively working on improving their tooling to extract compilation-time metrics[34].

We surveyed the *V8*[35] JavaScript and WebAssembly compiler, the *Clang*[36] compiler in LLVM, the *GCC*[37] compiler, the Java *HotSpot* [23] compiler, and the C# *RyuJIT*[38] compiler in order to identify to which extent their capabilities overlap with the ideas proposed by ICON. Table 2 shows our findings from analyzing the documentation and source code of the compilers, as well as information provided by online forums and mailing lists. We tried to find out whether the compilers provide a way of extracting compilation-time metrics (column 1), whether they support the narrowing of extraction scopes (column 2), and whether they try to optimize compilation time based on outliers (column 3).

Based on the available compiler source code and the presence of compiler flags described in the online documentation, all surveyed compilers provide compilation-time metrics, although to varying degrees. To the best of our knowledge, HotSpot provides the compilation time

[33] https://github.com/llvm/llvm-project/labels/slow-compile, https://gcc.gnu.org/pipermail/gcc-bugs/2024-March/857635.html, https://discourse.llvm.org/t/gsoc-2024-statistical-analysis-of-llvm-ir-compilation-with-clang/77532
[34] https://bugs.openjdk.org/browse/JDK-8311896
[35] https://v8.dev/
[36] https://clang.llvm.org/
[37] https://gcc.gnu.org/
[38] https://github.com/dotnet/runtime/blob/main/docs/design/coreclr/jit/ryujit-overview.md

Table 2 Capabilities of compilation-time optimization of V8, Clang, GCC, HotSpot, RyuJIT, and GraalVM with ICON.

Compiler	Metrics extraction	Scope narrowing	Outlier analysis
V8	yes	no	no
Clang	yes	yes	no
GCC	yes	yes	no
HotSpot	yes	no	no
RyuJIT	yes	no	no
GraalVM with ICON	yes	yes	yes

across all compilations via the `-XX:+CITime` flag. V8 and RyuJIT report a fixed set of metrics for all compilation units via compiler flags (V8[39], RyuJIT[40]). Clang[41] and GCC[42] provide detailed reports about time spent in individual optimizations via the `-ftime-report` flag.

Regarding scope narrowing, Clang allows passing a parameter to the `-ftime-report` compiler flag to differentiate whether the time is reported "per-pass" or "per-pass-run" to separate or combine individual pass executions. GCC supports narrowing via a separate compiler flag `-ftime-report-details`. As far as we know, none of the other compilers provide options to change the scope of extracted metrics.

To the best of our knowledge, the GraalVM compiler with our ICON enhancements is the only compiler using outliers to improve compilation time.

6.2 Synergy with Regression Testing

In addition to finding optimization opportunities in compilers, ICON is well suited to accompany compiler regression testing. While regression testing ensures that changes to the source code do not break existing compiler features and do not impair the correctness of the produced machine code [43], our approach ensures that changes to the source code do not lead to compilation-time regressions, additional memory allocation, or other aspects negatively impacting compilation. Therefore, in addition to ensuring the correctness of the output via regression testing, ICON ensures the efficiency of the compilation process and identifies any newly introduced defects. Both can execute the same tests, so no additional input programs are required to integrate ICON.

7 Conclusion

We presented ICON, a new data-driven approach to compilation-time optimization that splits high-level metrics into individual source program functions, compiler optimizations, or even into individual instruction in the compiler source code. By focusing on outliers in the extracted data, this approach can identify potential optimization opportunities in compiler implementations that are usually overlooked and provides a systematic approach to the analysis of compilation-time metrics.

[39] `https://chromium.googlesource.com/v8/v8/+/refs/heads/main/src/diagnostics/compilation-statistics.h,https://v8.dev/docs/trace`
[40] `https://github.com/dotnet/runtime/blob/main/src/coreclr/jit/compiler.h#L1643, https://github.com/dotnet/runtime/blob/main/docs/design/coreclr/jit/viewing-jit-dumps.md#miscellaneous-always-available-configuration-options`
[41] `https://releases.llvm.org/12.0.0/tools/clang/docs/ClangCommandLineReference.html#cmdoption-clang1-ftime-report`
[42] `https://gcc.gnu.org/onlinedocs/gcc/Developer-Options.html#index-ftime-report`

To demonstrate the effectiveness of our approach, we used ICON to extract a detailed view of the compilation time of the individual optimizations of the GraalVM compiler and performed a comprehensive outlier analysis on the resulting data with the goal of finding optimization potential. We found that most of the compilation time is spent in partial evaluation throughout all compiler tiers and editions, while the front end takes more time than the back end, especially in Tier-2 compilations.

The outlier analysis led to one language-agnostic and three language-specific outliers in compilation time. In the outlier analysis, the spatial component of the extraction-scope narrowing turned out to be useful at all levels. While the per-node analysis of partial evaluation helped us to identify an optimization opportunity in constant-folding, the per-phase analysis applied to the compiler mid tier, low tier and back end led to the detection of compiler defects in Python, JavaScript, and the GraalVM LLVM runtime, and helped us to develop an improved algorithm for the frame-state-assignment phase. Similarly, the temporal component was useful for analyzing Python, and we identified a single function as the compiler defect source.

Based on the identified optimization opportunities, we added additional caches to the constant-folding performed during partial evaluation and implemented a new deletion strategy for unused nodes in the frame-state-assignment phase. We reported the two remaining findings to the GraalVM compiler and language teams. The implemented optimizations improved compilation time in all languages between 2.25% (Python) and 9.45% (Espresso).

References

1 Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986. URL: `https://www.worldcat.org/oclc/12285707`.

2 Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. A survey on compiler autotuning using machine learning. *ACM Comput. Surv.*, 51(5):96:1–96:42, September 2019. `doi:10.1145/3197978`.

3 Islem Bouzenia and Michael Pradel. Resource usage and optimization opportunities in workflows of github actions. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pages 25:1–25:12, Los Alamitos, CA, USA, April 2024. ACM. `doi:10.1145/3597503.3623303`.

4 E. O. Brigham and R. E. Marrow. The fast fourier transform. *IEEE Spectrum*, 4(12):63–70, 1967. `doi:10.1109/MSPEC.1967.5217220`.

5 Michael D. Brown, Matthew Pruett, Robert Bigelow, Girish Mururu, and Santosh Pande. Not so fast: understanding and mitigating negative impacts of compiler optimizations on code reuse gadget sets. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–30, October 2021. `doi:10.1145/3485531`.

6 Timofey Bryksin, Victor Petukhov, Ilya Alexin, Stanislav Prikhodko, Alexey Shpilman, Vladimir Kovalenko, and Nikita Povarov. Using large-scale anomaly detection on code to improve kotlin compiler. In Sunghun Kim, Georgios Gousios, Sarah Nadi, and Joseph Hejderup, editors, *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*, MSR '20, pages 455–465, New York, NY, USA, 2020. ACM. `doi:10.1145/3379597.3387447`.

7 Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Trans. Program. Lang. Syst.*, 17(2):181–196, March 1995. `doi:10.1145/201059.201061`.

8 Cliff Click and Michael Paleczny. A simple graph-based intermediate representation. In Michael D. Ernst, editor, *Proceedings ACM SIGPLAN Workshop on Intermediate Representations (IR'95), San Francisco, CA, USA, January 22, 1995*, IR '95, pages 35–49, New York, NY, USA, 1995. ACM. `doi:10.1145/202529.202534`.

9 Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Mary S. Van Deusen and Bernard Lang, editors, *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*, POPL '93, pages 493–501, New York, NY, USA, 1993. ACM Press. doi:10.1145/158511.158707.

10 Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. Graal ir: An extensible declarative intermediate representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*, pages 1–9, 2013.

11 Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In Christoph Bockisch, Michael Haupt, Steve Blackburn, Hridesh Rajan, and Joseph Gil, editors, *VMIL@SPLASH '13: Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages, Indianapolis, IN, USA, 28 October 2013*, VMIL '13, pages 1–10, New York, NY, USA, 2013. ACM. doi:10.1145/2542142.2542143.

12 Gilles Marie Duboscq. Combining speculative optimizations with flexible scheduling of side-effects, 2016. URL: https://resolver.obvsg.at/urn:nbn:at:at-ubl:1-9708.

13 Josef Eisl. Trace register allocation, 2018. URL: https://resolver.obvsg.at/urn:nbn:at:at-ubl:1-25787.

14 Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, François Bodin, Phil Barnard, Elton Ashton, Edwin V. Bonilla, John Thomson, Christopher K. I. Williams, and Michael F. P. O'Boyle. Milepost GCC: machine learning enabled self-tuning compiler. *Int. J. Parallel Program.*, 39(3):296–327, June 2011. doi:10.1007/s10766-010-0161-2.

15 Yoshihiko Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *High. Order Symb. Comput.*, 12(4):381–391, December 1999. doi:10.1023/A:1010095604496.

16 G. Genta G. Barbato, E. M. Barini and R. Levi. Features and performance of some outlier detection methods. *Journal of Applied Statistics*, 38(10):2133–2149, 2011. doi:10.1080/02664763.2010.545119.

17 James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. The java® language specification, 2024. URL: https://docs.oracle.com/javase/specs/jls/se22/jls22.pdf.

18 Tobias Hartmann, Albert Noll, and Thomas R. Gross. Efficient code management for dynamic multi-tiered compilation systems. In Joanna Kolodziej and Bruce R. Childers, editors, *2014 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ '14, Cracow, Poland, September 23-26, 2014*, PPPJ '14, pages 51–62, New York, NY, USA, 2014. ACM. doi:10.1145/2647508.2647513.

19 Urs Hölzle, Craig Chambers, and David M. Ungar. Debugging optimized code with dynamic deoptimization. In Stuart I. Feldman and Richard L. Wexelblat, editors, *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI), San Francisco, California, USA, June 17-19, 1992*, volume 27, pages 32–43, New York, NY, USA, July 1992. ACM. doi:10.1145/143095.143114.

20 Kenneth Hoste and Lieven Eeckhout. Cole: compiler optimization level exploration. In Mary Lou Soffa and Evelyn Duesterwald, editors, *Sixth International Symposium on Code Generation and Optimization (CGO 2008), April 5-9, 2008, Boston, MA, USA*, CGO '08, pages 165–174, New York, NY, USA, 2008. ACM. doi:10.1145/1356058.1356080.

21 Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. A domain-specific language for building self-optimizing AST interpreters. In Ulrik Pagh Schultz and Matthew Flatt, editors, *Generative Programming: Concepts and Experiences, GPCE'14, Vasteras, Sweden, September 15-16, 2014*, volume 50, pages 123–132, New York, NY, USA, September 2014. ACM. doi:10.1145/2658761.2658776.

22 Tarindu Jayatilaka, Hideto Ueno, Giorgis Georgakoudis, Eunjung Park, and Johannes Doerfert. Towards compile-time-reducing compiler optimization selection via machine learning. In Federico Silla and Osni Marques, editors, *ICPP Workshops 2021: 50th International Conference on Parallel Processing, Virtual Event / Lemont (near Chicago), IL, USA, August 9-12, 2021*, ICPP Workshops '21, pages 23:1–23:6, New York, NY, USA, 2021. ACM. doi:10.1145/3458744.3473355.

23 Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth B. Russell, and David Cox. Design of the java hotspot™ client compiler for java 6. *ACM Trans. Archit. Code Optim.*, 5(1):7:1–7:32, May 2008. doi:10.1145/1369396.1370017.

24 Sameer Kulkarni and John Cavazos. Mitigating the compiler optimization phase-ordering problem using machine learning. In Gary T. Leavens and Matthew B. Dwyer, editors, *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, OOPSLA '12, pages 147–162, New York, NY, USA, 2012. ACM. doi:10.1145/2384616.2384628.

25 Octave Larose, Sophie Kaleba, Humphrey Burchell, and Stefan Marr. AST vs. bytecode: Interpreters in the age of meta-compilation. *Proc. ACM Program. Lang.*, 7(OOPSLA2):318–346, October 2023. doi:10.1145/3622808.

26 Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, and Daniel Smith. The java® virtual machine specification, 2024. URL: https://docs.oracle.com/javase/specs/jvms/se22/jvms22.pdf.

27 Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. Cross-language compiler benchmarking: are we fast yet? In Roberto Ierusalimschy, editor, *Proceedings of the 12th Symposium on Dynamic Languages, DLS 2016, Amsterdam, The Netherlands, November 1, 2016*, volume 52, pages 120–131, New York, NY, USA, November 2016. ACM. doi:10.1145/2989225.2989232.

28 Uwe Meyer. Techniques for partial evaluation of imperative languages. In Charles Consel and Olivier Danvy, editors, *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'91, Yale University, New Haven, Connecticut, USA, June 17-19, 1991*, PEPM '91, pages 94–105, New York, NY, USA, 1991. ACM. doi:10.1145/115865.115876.

29 Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

30 Gennady Pekhimenko and Angela Demke Brown. Efficient program compilation through machine learning techniques. In Ken Naono, Keita Teranishi, John Cavazos, and Reiji Suda, editors, *Software Automatic Tuning, From Concepts to State-of-the-Art Results*, pages 335–351. Springer, New York, NY, 2010. doi:10.1007/978-1-4419-6935-4_19.

31 Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, June 1975. doi:10.1145/360825.360839.

32 Dmitry Plotnikov, Dmitry Melnik, Mamikon Vardanyan, Ruben Buchatskiy, Roman Zhuykov, and Je-Hyung Lee. Automatic tuning of compiler optimizations and analysis of their impact. In Vassil Alexandrov, Michael Lees, Valeria V. Krzhizhanovskaya, Jack J. Dongarra, and Peter M. A. Sloot, editors, *Proceedings of the International Conference on Computational Science, ICCS 2013, Barcelona, Spain, 5-7 June, 2013*, volume 18 of *Procedia Computer Science*, pages 1312–1321. Elsevier, 2013. 2013 International Conference on Computational Science. doi:10.1016/j.procs.2013.05.298.

33 Manuel Rigger, Matthias Grimmer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. Bringing low-level languages to the JVM: efficient execution of LLVM IR on truffle. In Antony L. Hosking and Witawas Srisa-an, editors, *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages, VMIL@SPLASH 2016, Amsterdam, The Netherlands, October 31, 2016*, VMIL 2016, pages 6–15, New York, NY, USA, 2016. ACM. doi:10.1145/2998415.2998416.

34 Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Icebreaker: warming serverless functions better with heterogeneity. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, editors, *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, ASPLOS '22, pages 753–767, New York, NY, USA, 2022. ACM. doi:10.1145/3503222.3507750.

35 Doug Simon, Christian Wimmer, Bernhard Urban, Gilles Duboscq, Lukas Stadler, and Thomas Würthinger. Snippets: Taking the high road to a low level. *ACM Trans. Archit. Code Optim.*, 12(2):20:20:1–20:20:25, June 2015. doi:10.1145/2764907.

36 Matija Sipek, Branko Mihaljevic, and Aleksander Radovan. Exploring aspects of polyglot high-performance virtual machine graalvm. In Marko Koricic, Zeljko Butkovic, Karolj Skala, Zeljka Car, Marina Cicin-Sain, Snjezana Babic, Vlado Sruk, Dejan Skvorc, Slobodan Ribaric, Stjepan Gros, Boris Vrdoljak, Mladen Mauher, Edvard Tijan, Predrag Pale, Darko Huljenic, Tihana Galinac Grbac, and Matej Janjic, editors, *42nd International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2019, Opatija, Croatia, May 20-24, 2019*, pages 1671–1676. IEEE, 2019. doi:10.23919/MIPRO.2019.8756917.

37 Matija Sipek, D. Muharemagic, Branko Mihaljevic, and Aleksander Radovan. Enhancing performance of cloud-based software applications with graalvm and quarkus. In Marko Koricic, Karolj Skala, Zeljka Car, Marina Cicin-Sain, Vlado Sruk, Dejan Skvorc, Slobodan Ribaric, Bojan Jerbic, Stjepan Gros, Boris Vrdoljak, Mladen Mauher, Edvard Tijan, Tihomir Katulic, Predrag Pale, Tihana Galinac Grbac, Nikola Filip Fijan, Adrian Boukalov, Dragan Cisic, and Vera Gradisnik, editors, *43rd International Convention on Information, Communication and Electronic Technology, MIPRO 2020, Opatija, Croatia, September 28 - October 2, 2020*, pages 1746–1751. IEEE, 2020. doi:10.23919/MIPRO48935.2020.9245290.

38 Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. Partial escape analysis and scalar replacement for java. In David R. Kaeli and Tipp Moseley, editors, *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2014, Orlando, FL, USA, February 15-19, 2014*, CGO '14, page 165, New York, NY, USA, 2014. ACM. doi:10.1145/2581122.2544157.

39 James Stanier and Des Watson. Intermediate representations in imperative compilers: A survey. *ACM Comput. Surv.*, 45(3):26:1–26:27, July 2013. doi:10.1145/2480741.2480743.

40 Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, April 1991. doi:10.1145/103135.103136.

41 Christian Wimmer. Linear scan register allocation for the java hotspot™ client compiler, 2004.

42 Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. Initialize once, start fast: application initialization at build time. *Proc. ACM Program. Lang.*, 3(OOPSLA):184:1–184:29, October 2019. doi:10.1145/3360610.

43 W. Eric Wong, Joseph R. Horgan, Saul London, and Hiralal Agrawal. A study of effective regression testing in practice. In *Eighth International Symposium on Software Reliability Engineering, ISSRE 1997, Albuquerque, NM, USA, November 2-5, 1997*, pages 264–274. IEEE Computer Society, November 1997. doi:10.1109/ISSRE.1997.630875.

44 Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, PLDI 2017, pages 662–676, New York, NY, USA, 2017. ACM. doi:10.1145/3062341.3062381.

45 Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to rule them all. In Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld, editors, *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, Onward! 2013, pages 187–204, New York, NY, USA, 2013. ACM. doi:10.1145/2509578.2509581.

46 Yifei Zhang, Tianxiao Gu, Xiaolin Zheng, Lei Yu, Wei Kuai, and Sanhong Li. Towards a serverless java runtime. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*, pages 1156–1160. IEEE, 2021. doi:10.1109/ASE51524.2021.9678709.

A Frame-State Algorithm

Listing 3 Pseudo-code representation of the updated deletion algorithm.

```
void deleteUnusedNodes(List<Node> deleteList) {
    // track the usages of each node
    Map<Node, Int> usages = new Map();
    Set<Node> maybeDelete = new Set();

    // delete the initial set of nodes
    for (Node n in deleteList) {
        delete(n);
        for (Node input in n.inputs()) {
            Int u = usages[input];
            if (u == null) {
                u = input.usageCount();
            }
            usages[input] = u - 1;
            maybeDelete.add(input);
        }
    }

    // fixed point iteration to delete nodes transitively
    for (Node n in maybeDelete) {
        if (shouldBeDeleted(n, u)) {
            delete(n);
            for (Node input in n.inputs()) {
                Int u = usages[input];
                if (u == null) {
                    u = input.usageCount();
                }
                usages[input] = u - 1;
                maybeDelete.add(input);
            }
        }
    }

    // remove the usages of nodes that were not deleted
    // and for which the usage count changed
    for (Node n, Int u in usages) {
        if (isAlive(n) && n.usageCount() != u) {
            n.removeDeadUsages();
        }
    }
}
```