

Author
DI **Christoph Pichler**, BSc
01605053

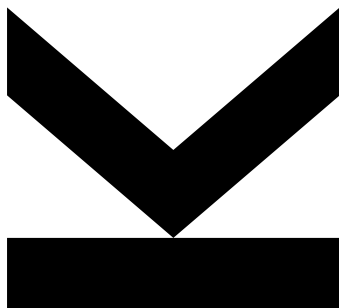
Submission
SSW | Institute for
System Software

First Supervisor
Univ.-Prof. Dr. **Stefan Marr**

Second Supervisor
Univ.-Prof. Dr. **Daniel Große**

July 2026

Mixing Execution Modes to Balance Trade-offs in Modern Runtimes



Doctoral Thesis

to confer the academic degree of

Doktor der technischen Wissenschaften

in the Doctoral Program

Technische Wissenschaften

Abstract

Modern runtimes face fundamental trade-offs between peak performance, warm-up performance, and application size: AOT compilation enables fast warm-up by pre-compiling code. JIT compilation dynamically optimizes hot code paths to achieve high peak performance, while cold paths are executed in a less-optimized mode or even interpreted. However, AOT compilation typically yields large binaries, while profiling and optimizing at run time introduce warm-up costs for JIT compilation. These trade-offs can be mitigated by hybrid execution models, which combine multiple execution modes. While execution modes are specific to the language implementation, some systems enable mixed-mode execution, for example, via native extensions. However, this approach requires developers to explicitly manage mode selection by strategically choosing language implementations for different code segments.

This work, in contrast, shows that automatically mixing execution modes unlocks missed optimization possibilities. On the one hand, it is demonstrated how to reduce binary executable size: Rarely executed code is not compiled to highly optimized machine code, but can instead be kept as compact bytecode, executed by a lightweight interpreter. On the other hand, this work shows that mixing execution modes can achieve peak and warm-up performance benefits.

Evaluations on DaCapo, Renaissance, lxml and ujson benchmarks show that it can pay off to automatically mix execution modes. While static heuristics are a feasible choice and can be applied quickly, dynamic heuristics achieve better results in most cases. When heuristics are developed and tuned, it is important to consider code that is by design bound to run in a certain mode. Also, experiments indicate that the characteristic properties of initialization code affect performance and code size in a different way than other code does.

This work has been implemented on top of GraalVM, a high-performance, production-grade and multi-language runtime that is also widely used in industry. This demonstrates the feasibility of this work in state-of-the-art systems and shows its impact on an already highly optimized environment.

Kurzfassung

Moderne Laufzeitumgebungen stehen vor grundlegenden Kompromissen zwischen Höchstleistung, Dauer der Aufwärmphase und Größe des Programms: Ahead-of-Time-Kompilierung ermöglicht einen schnellen Start durch das vorzeitige Kompilieren des Codes. Just-in-Time-Kompilierung optimiert häufig genutzte Codepfade zur Laufzeit, um eine gute Höchstleistung zu erzielen. Selten genutzte Pfade werden hingegen weniger optimiert ausgeführt oder sogar interpretiert. Allerdings führt eine vorzeitige Kompilierung in der Regel zu großen Binärdateien, während die Analyse und die Optimierung zur Laufzeit Aufwärmkosten für die JIT-Kompilierung verursachen. Diese Kompromisse lassen sich durch hybride Ausführungsmodelle, die mehrere Ausführungsarten kombinieren, abschwächen. Da die Implementierung einer Programmiersprache eines Codes den Ausführungsmodus bestimmt, ermöglichen manche Systeme eine Ausführung mit gemischten Modi, etwa über Native Extensions. Dieser Ansatz erfordert jedoch, dass Entwickler:innen die Implementierungen der Programmiersprachen für verschiedene Codesegmente strategisch explizit auswählen, um die entsprechenden Ausführungsmodi festzulegen.

Diese Arbeit zeigt hingegen, dass das automatisierte Kombinieren verschiedener Ausführungsmodi neues Verbesserungspotential ermöglicht. Einerseits wird gezeigt, wie sich die Größe der kompilierten Programmdateien reduzieren lässt: Selten ausgeführter Code wird nicht zu hochoptimiertem Maschinencode kompiliert, sondern kann stattdessen als kompakter Bytecode behalten und von einem einfachen Interpreter ausgeführt werden. Andererseits zeigt diese Arbeit, dass Programme durch das Kombinieren verschiedener Ausführungsmodi Verbesserungen bezüglich Höchstleistung und Aufwärmphasen erzielen können.

Auswertungen anhand von DaCapo-, Renaissance-, lxml- und ujson-Benchmarks zeigen, dass es sich lohnen kann, Ausführungsmodi automatisiert zu kombinieren. Während statische Heuristiken eine praktische Möglichkeit darstellen und sich schnell anwenden lassen, erzielen dynamische Heuristiken in der Regel bessere Ergebnisse. Beim Entwickeln und Optimieren von Heuristiken ist es wichtig, Code zu berücksichtigen, der zwangsläufig in einem bestimmten Modus ausgeführt werden muss. Außerdem zeigen Experimente, dass die charakteristischen Eigenschaften von Initialisierungscode die Leistung und die Codegröße anders als durchschnittlicher Code beeinflussen.

Die Implementierungen dieser Arbeit basieren auf GraalVM, einer leistungsstarken und mehrsprachigen Laufzeitumgebung, die auch in der Industrie weit verbreitet ist. Dies zeigt die Umsetzbarkeit dieser Arbeit in modernen Systemen des aktuellen Stands, und verdeutlicht, dass trotz bereits hochoptimierter Systeme mit den Ansätzen dieser Arbeit noch Verbesserungen erreicht werden können.

Acknowledgements

Writing the final lines of this thesis and reaching a state where my PhD studies are about to be finished, after these four years, feels surreal. As part of this PhD, I have been researching, experimenting, drafting and discarding plans, making mistakes, and learning. None of this would have been possible, however, without the support of those around me. Even though I am the author of this thesis, numerous people were involved – more than I can name here, and I am sorry for not mentioning all of them here. However, to all of them, I would like to express my immense gratitude. No matter whether it was support with my research or support that helped me become the person I am today.

My heartfelt thanks go to Hanspeter Mössenböck. He served as the primary advisor for this thesis for three of the four years and, prior to that, helped me get started in compiler research at the SSW during my master's program. The question "What would Peter do?" remains a perfect guiding principle when it comes to effective presentation skills and concise, efficient writing. He also provided general support to students, particularly regarding study abroad semesters. Thanks to him, I was able to make one of the best decisions of my studies and spend a semester studying at TU Delft. I would also like to generally thank his successor, Stefan Marr, who has now become my primary advisor in the final year. In particular, his critical questions about my own research, followed by helpful tips, have significantly raised the level of my work and improved the research results. Last but not least, I would like to thank my secondary advisor, Daniel Große, as well as Christos Kotselidis, who agreed to review this thesis.

This research was the result of a collaboration with Oracle. I would therefore like to take this opportunity to express my gratitude, especially to Roland Schatz, who has been a constant source of guidance since my bachelor's thesis and always has a solution at the ready. I would also like to thank Bernhard Urban-Forster for his support in the second half of my PhD. A big thank you goes to Paley Li: He not only provided technical and subject-matter support, but also gave great feedback. Even though I regularly tested his patience, he consistently took the time – even after leaving Oracle – to support me throughout the PhD process until the very end.

Also, this work would not have been possible without my colleagues – from professional feedback to organizational support in research and teaching, and also the informal discussions besides our actual work. Special thanks go to Jacob Kreindl, who not only supervised my bachelor's thesis but also, as a colleague in the office, was always there to answer questions about Sulong. I would also like to extend my heartfelt thanks to Dominik, Michi H., Lukas, Sebastian, Raphael, Markus, Herbert, Christoph, and Thomas. And of course Karin and Michi K., who – though not always visible for people outside of SSW – have reliably been keeping the institute and thus our research and teaching running.

Also, I would like to thank Jakob and HPU for our mutual support during the Bachelor's courses of computer science. And even earlier, all who supported my computer science and math interest, specifically Markus Oppitz and Gertrude Eder.

My heartfelt thanks also go to my parents. They have supported me my entire life and have always stood by me. Without them, none of this would have been possible. In general, I would also like to thank everyone who has supported me outside of my studies and stood by me. That includes not only my siblings Verena and Alex, but also the "Wunderkinder", who carried me through some very difficult times in my life. I have also found and received support and friendship through many clubs – Quadball, TSC Wechselschritt, and HOSI Linz. To this day, music remains my faithful counterbalance to my research. So I would also like to thank the "Wunderkinder", Celebrate Music, and everyone else who has supported me on my musical journey. Finally, I would like to give special thanks to Lukas. Although I have not met him until the final phase of my PhD, it is precisely now that he is giving me stability without making me compromise who I am.

Contents

Abstract	iii
Kurzfassung	v
Acknowledgements	vii
1 Introduction	1
1.1 Problems	2
1.2 Goals	4
1.3 Approach	4
1.4 Supporting Publications	5
1.5 Outlook and Outline	7
2 Background	9
2.1 Program Compilation and Execution	9
2.1.1 Ahead-of-Time Compilers	9
2.1.2 Interpretation	10
2.1.3 Just-in-Time Compilation	11
2.1.4 Properties of Execution Modes	12
2.1.5 Native Extensions	12
2.2 Compiler Optimizations	13
2.2.1 Inlining	13
2.2.2 Profile-Guided Optimization	14
2.3 Compiler Heuristics	14
2.3.1 Foundational Concepts	15
2.3.2 Optimizations Along the Trade-off Between Size and Performance	15
2.3.3 Combining Different Optimization Levels	16
2.4 Linking and Lazy Function Resolution	16
2.5 Selected Language Implementations	18
2.5.1 LLVM	18
2.5.2 Python and its Implementations	19
2.5.3 Java Virtual Machine	19
2.6 GraalVM	20
2.6.1 GraalVM Compiler	21
2.6.2 Truffle Language Implementation Framework	21
2.6.3 Sulong	22
2.6.4 Truffle Interoperability	22
2.6.5 Truffle Native Function Interface	23
2.6.6 Sulong Toolchain	24
2.6.7 GraalPy	25
2.6.8 Native Image	25
2.7 Statistics	28
2.7.1 Box Plots	28

2.7.2	Correlation Coefficients	29
2.8	Convention of the Terms Runtime, Run Time and Run-Time	29
3	Mixing Execution Modes to Reduce Binary Size	31
3.1	Motivation	31
3.2	Code Features and Correlations	34
3.2.1	Methodology	34
3.2.2	Run-Time Data: Execution Time Ratio	34
3.2.3	Correlations of Static Features and Execution Time Ratio . .	35
3.2.4	Static Initializers	40
3.3	Research Questions	42
3.4	Heuristics	43
3.4.1	Static Decision: Per-Compilation-Unit Compile Score	43
3.4.2	Single-Feature Static Heuristics	44
3.4.3	Combined Static Heuristics	45
3.4.4	Dynamic Heuristic	46
3.5	Implementation	47
3.5.1	GraalVM Native Image	47
3.5.2	Triggering Interpretation of Compilation Units from Native Call Sites	48
3.5.3	Execution Mode Restrictions	49
3.5.4	Performance-Critical Compilation Units	49
3.5.5	Imperfection of Heuristics: Need for Run-Time Compilation	49
3.5.6	Constant Overhead: Sizes of Interpreter and JIT Compiler . .	51
3.6	Evaluation	52
3.6.1	Methodology	53
3.6.2	Results for Single-Feature Heuristics (research question 1) .	54
3.6.3	Results for Combined Static Heuristics (research question 1)	54
3.6.4	Results for Dynamic Heuristics With and Without Simu- lated JIT Compilation (Research Question 2)	58
3.6.5	Comparing Static and Dynamic Results (Research Question 3)	63
3.7	Limitations	66
3.8	Summary	66
4	Implementing Hybrid Execution of Native Extensions in a Multi-Language VM	69
4.1	Motivation	69
4.2	Basic Mixing Approach	71
4.3	Heap Access	71
4.3.1	Language Semantics Violation: Native Code Accessing Managed Objects	71
4.3.2	Access Scenarios	72
4.4	Native Symbol Resolution	76
4.4.1	Retrieving Symbol Information	76
4.4.2	Library Loading and Symbol Resolution	76
4.4.3	Resolution of Native Static Symbols	77
4.5	Switching Execution Mode: Method and Function Calls	78
4.5.1	Calling Native Functions From Managed Code	79
4.5.2	Native Execution: Limitations	79
4.5.3	Calling Managed Functions From Native Code	80
4.6	Foreign Caller Calls Native Callee	82

4.7	Limitations	82
4.8	Summary	83
5	Heuristics for Mixing Execution Modes in Native Extensions	85
5.1	Motivation	85
5.2	Function Execution Time in Different Execution Modes	86
5.2.1	Execution Time Estimation for Single Functions	86
5.2.2	Function Call Overhead of a Single Function	89
5.3	Heuristics	91
5.4	Evaluation	92
5.4.1	lxml Benchmarks	92
5.4.2	UltraJSON Benchmarks	95
5.5	Discussion	96
5.6	Summary	97
6	Related Work	99
6.1	Technical Concepts	99
6.2	Compiler Heuristics	100
6.2.1	Adaptive and Dynamic Compiler Heuristics	100
6.2.2	Static Branch Prediction	101
6.2.3	Optimizations Along the Trade-off Between Size and Performance	101
6.2.4	Optimization Selection	102
7	Future Work	105
7.1	Technical Implementation of Hybrid Execution	105
7.2	Heuristics	106
8	Conclusion	109
	List of Figures	111
	List of Tables	115
	Listings	117
	List of Acronyms	119
	Bibliography	121

Chapter 1

Introduction

A compiler is a tool that generates machine code from source code. This translation process can happen at various times: Ahead of time (AOT) compilers transform the source code to machine code prior to any execution. Interpreters, in contrast, execute a program without first compiling it to machine code [63].

For a long time, AOT compilation has been the standard compilation technique, producing binaries with instant warm-up [3, 63]. For dynamically typed languages, however, certain optimizations are difficult to perform AOT, since the behavior of the program may depend on run-time data.

Therefore, just in time (JIT) compilation has been developed to improve peak performance: Since a JIT compiler optimizes the program during an ongoing execution, it has access to run-time information such as actual data types, branch probabilities, and frequently used code [48]. Thus, it can produce highly optimized machine code that a static compiler might not be able to generate. However, JIT compilation usually comes with longer and unpredictable warm-up times [3, 7] resulting from run-time optimization. Although there are approaches to reduce warm-up times [4, 59], these systems cannot compete with the instant warm-up of AOT-compiled code.

So, AOT and JIT compilation as well as interpretation can be considered as different execution modes, each coming with different advantages and disadvantages. Thus, modern runtimes encounter fundamental trade-offs among peak performance, warm-up performance, and application size. Figure 1.1 shows AOT compilation, JIT compilation and interpretation as the corners of a triangle. Mixing two of these execution modes usually leads to the advantage listed on the edge, while leading to the trade-off between the other two edges.

As an example, the Java HotSpot™ VM starts with interpretation and offers different levels of JIT compilation [64]. This keeps file sizes low and tries to balance a fast warm-up, while still reaching high peak performance eventually. Still, it is hard to achieve both at high level, a fast warm-up as well as high peak performance. A recent possibility for that came up with project Leyden [83], where profiling runs are performed ahead of time. However, this requires effort for profiling.

As another example, optimizing AOT compilers such as GraalVM Native Image produces executables with fast warm-up and high peak performance [110]. However, this comes with the disadvantage that the size of resulting binary executables are significantly larger than their intermediate representation format used for interpretation.

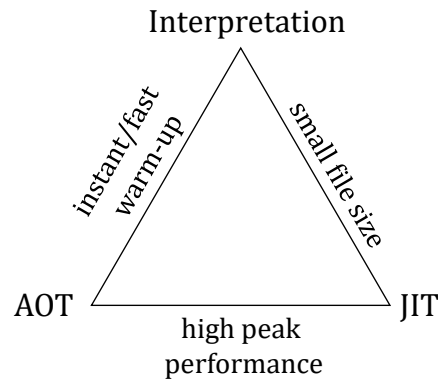


Figure 1.1: Three execution modes (AOT compilation, JIT compilation and interpretation) and their advantages.

By and large, we are envisioning a system that mixes different execution modes to balance the trade-offs mentioned above. Such a system should automatically decide how to execute certain code, which requires that the same source code can be executed in different modes.

1.1 Problems

The various drawbacks of different execution modes can lead to the following problems:

Large Binary Executables of AOT-Compiled Code Ahead-of-time executables are ready-to-execute applications. This means, they need to have everything included that is necessary at run time. Thus, they also contain code that is never or only rarely executed, which leads to large binary files.

Achieving both fast warm-up and high peak performance simultaneously is a known difficulty [110]. One solution is an extensive code analysis followed by highly aggressive optimization and compilation ahead of time (as a JIT compiler would introduce warm-up again). However, that extensive analysis, optimization and AOT compilation, is also applied to rarely or never executed code. This does not only increase compilation effort, but also typically increases the size of the resulting binary executable even more.

Warm-up Times of JIT-Compiled Code JIT compilation, which optimizes hot code paths for high peak performance, introduces warm-up costs due to profiling and run-time optimization. Especially for continuous deployment, where services are restarted up to 30 times per day, high warm-up costs negatively affects the overall performance of applications [27, 28].

Mixing Execution Modes: Possible, but not Trivial To combine the respective advantages of different execution modes, systems have arisen that allow combining multiple languages [36, 64]. Popular use cases are native extensions:

Computationally intensive code is written as a native extension in a lower-level language such as C and compiled AOT. In contrast, the remainder of the application is written in a higher-level language like Python. This combines the warm-up advantages of native execution together with the flexibility of higher-level languages.

However, mixing execution modes in such a way requires a mechanism to change execution mode during ongoing execution. Depending on the implementation, either all modes need to use the same memory model. Otherwise, it is necessary to convert between different memory representations as soon as the same heap object has to be accessed from different execution modes.

Programmers Manually Selecting Execution Modes An approach like native extensions still requires the programmer to decide the execution mode manually by selecting in which language to write the code. However, such a manual decision how to execute code requires the programmer to be familiar with the system and to have compiler knowledge in general.

Additionally, certain native extensions are used in libraries such as numpy for Python [67]. As their code is used in different projects with different contexts, it is impossible for the programmer of the library to optimize for all future usages of such a library. Therefore, it is still possible to miss optimization opportunities even for compiler experts.

Mixing Execution Modes May Bring up Existing Problems of AOT-/JIT-Compiled Code To avoid disadvantages such as high warm-up times, one can start to transition execution from one mode to another over time. As an example, a JIT compiler can cache compilation results and reuse it in future runs [59]. This reduces warm-up costs of future runs on the one hand. On the other hand, it has the drawback of not avoiding warm-up work for early iterations. Also, eventually, every executed code is compiled to machine code again, which leads to the existing problem of large binary files. Thus, mixing execution modes does not automatically remove drawbacks of the individual modes.

Constraints for Automatic Mixing In a system that automatically mixes execution modes, the selection which mode to use for certain code can be taken at different points within the compilation toolchain. It is not trivial to decide where this point should be: An earlier decision means that less information is available, such that the overall decisions are of lower accuracy. A later decision, in contrast, might lead to better choices. However, later decisions can cause a higher compilation effort, as assumptions about an execution mode cannot be made for earlier parts of the compilation toolchain, which leaves less opportunities for optimization.

Additionally, there can be some parts within the application that cannot run in a certain mode, especially runtime code of managed execution mode such as an interpreter, a JIT compiler, or a garbage collector (GC). Such code parts have to be excluded from analysis. However, there are systems that merge application and runtime code to apply global optimizations [110]. In such systems, such a separation is hard to do, if possible at all.

1.2 Goals

In general, we want to mix execution modes in order to combine the different advantages of various execution modes, and to tackle the problems listed above. As objectives, we want to keep the properties of the execution mode triangle in mind: Warm-up, peak performance, and binary application size. More concretely, our research has the following goals:

Goal 1: Reduce Size of Executables While Keeping Performance As mentioned, AOT-compiled executables offer both, a fast start-up as well as high peak performance. However, our goal is to keep these properties as good as possible, while reducing the file size at the same time.

Goal 2: Improve Performance of Manually Selected Execution Modes As written above, native extensions require a manual decision in which language to implement each part of the application. This can lead to suboptimal decisions by programmers. Thus, our goal is to find previously missed optimizations in applications with native extensions and improve performance by mixing execution modes within code of native extensions as well.

Goal 3: Automating the Selection of Execution Modes For both goals listed above, it is necessary to select an execution mode for every code segment. However, manual decisions might miss optimizations. Thus, our goal is to have an appropriate execution mode selected automatically for specific code segments. Of course, the selection should consider the mentioned objectives high peak performance, low warm-up time, or small binary executable size.

1.3 Approach

To achieve the goals mentioned above, we see two central approaches to be necessary: First, one needs to solve existing problems of a hybrid execution model itself. This includes the way of the different modes and the interaction between them within the runtime system. Second, it is necessary to identify and evaluate heuristics to automatically mix these execution modes, as proposed by goal 3: Instead of manual intervention, the system has to automatically determine and apply the most suitable execution mode for different parts of the code. For that, it is necessary for which objectives to optimize: While warm-up performance is important for rather short-running applications, peak performance might be more important than a fast warm-up for a long-running server application. Therefore, the optimization goal always depends on the context.

Concretely, we propose approaches for two different scenarios:

Reducing Size of Binary Executables To shrink the size of an AOT-compiled executable and achieve goal 1, we propose a prototype that enables mixing AOT compilation and interpretation. While AOT compilation guarantees fast execution, a lightweight interpreter enables rarely or never executed code to be run without having to be fully compiled to large binaries. To automatically select an appropriate execution mode and achieve goal 3, we propose static heuristics based on properties of the corresponding code. As a check what is possible, we can compare those results to a mode selection based on run-time profiling data. This approach is fully discussed in Chapter 3.

Combining Execution Modes in a Multi-Language Application To achieve goal 2, we propose a hybrid execution environment for native extensions in multi-language applications, enabling different execution modes within a single language. This also includes solutions for heap access and memory object handling to ensure correct interaction between accesses from different execution modes. The general hybrid execution model for native extensions, including the technical challenges and solutions, is discussed in Chapter 4.

To also apply automated heuristics and achieve goal 3, we propose to analyse the corresponding code and design heuristics for automatically selecting the execution mode for functions in native extensions. This is discussed in Chapter 5.

1.4 Supporting Publications

In the following, we list supporting publications of this thesis. The publications are sorted by the approaches defined above.

Reducing Size of Binary Executables Concerning the approach of mixing execution modes to reduce the file size of binary executables, we published the following two papers:

- [80] Christoph Pichler et al. 2025. Fast and Compact: Reducing Size of AOT-Compiled Java Code without Sacrificing Performance. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '25)*. Association for Computing Machinery, Singapore, Singapore, pp. 12–22. ISBN: 9798400721496. DOI: 10.1145/3759426.3760976
- [79] Christoph Pichler et al. 2026. To Compile or Not To Compile: Evaluating Static Heuristics to Reduce Binary Size of Hybrid Execution Systems. In *Proceedings of the 23rd ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '26)*. Association for Computing Machinery, Brussels, Belgium. DOI: 10.1145/3828170.3828171

While the first one [80] covers the technical implementation and uses dynamic heuristics to mix execution modes, the more recent one [79] analyses static heuristics.

Combining Execution Modes in a Multi-Language Application We first proposed the idea of hybrid execution for native extensions at the student research competition at the International Symposium on Code Generation and Optimization (CGO):

- [77] Christoph Pichler et al. 2023. Interoperability between Execution Modes on GraalVM. In *ACM Student Research Competition (SRC) at IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. <https://ssw.jku.at/Research/Papers/Pichler23/Pichler23SRC.pdf>

Then, we investigated what such an approach requires concerning the technical implementation. The approach itself and a proof of concept to show its performance improvements has been published in:

- [76] Christoph Pichler et al. 2023. Hybrid Execution: Combining Ahead-of-Time and Just-in-Time Compilation. In *Proceedings of the 15th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL 2023)*. Association for Computing Machinery, Cascais, Portugal, pp. 39–49. ISBN: 9798400704017. DOI: 10.1145/3623507.3623554

Afterwards, we enhanced our system by extending the flexibility of possible execution modes to choose. We analyzed different features and properties of different execution modes and their implementations, and found and evaluated heuristics that optimize for performance. This has been published in:

- [78] Christoph Pichler et al. 2024. On Automating Hybrid Execution of Ahead-of-Time and Just-in-Time Compiled Code. In *Proceedings of the 16th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '24)*. Association for Computing Machinery, Pasadena, CA, USA, pp. 1–11. ISBN: 9798400712135. DOI: 10.1145/3689490.3690398

Non-Dissertation Contributions I also made a peer-reviewed publication outside the scope of this thesis. In quantum computing, computation results often have a probability below 100% of being correct. Error-correcting codes increase this probability. Concretely, an input program is modified in a way such that wrong computation results correct themselves automatically in the output program, as long as the error probability does not exceed a certain threshold. The following publication introduces a framework for applying and evaluating error-correcting codes:

- [39] Thomas Grurl, Christoph Pichler, Jürgen Fuß, Robert Wille. 2023. Automatic Implementation and Evaluation of Error-Correcting Codes for Quantum Computing: An Open-Source Framework for Quantum Error Correction. In *2023 36th International Conference on VLSI Design and 2023 22nd International Conference on Embedded Systems (VLSID)*, pp. 301–306. DOI: 10.1109/VLSID57277.2023.00068

I contributed the implementation of automatically applying error-correcting mechanisms to a given quantum program within the framework.

1.5 Outlook and Outline

Overall, we demonstrate that automatically mixing execution modes within a modern, high-performance runtime enables previously missed optimization possibilities for balancing peak performance, warm-up latency, and binary executable size. By implementing and evaluating both static and dynamic heuristics to automatically select a suitable execution mode for specific code segments, we provide an approach that surpasses the limitations of manually mixing execution modes as well as the limitations of the individual execution modes themselves.

This thesis is structured as follows: Chapter 2 provides the necessary background for compilation and its heuristics in general and discusses Java, Python, LLVM, and the execution models of these languages. Also, it introduces GraalVM, which is a polyglot execution ecosystem. As shortly mentioned in Section 1.3, Chapter 3 describes our research about using hybrid execution to reduce binary size of AOT-compiled executables while keeping peak performance. While Chapter 4 covers necessary technical details for hybrid execution of native extensions, Chapter 5 is about heuristics to improve performance of systems described in Chapter 4. Then, Chapters 6 and 7 describe related and future work, respectively. Finally, Chapter 8 concludes this thesis.

Chapter 2

Background

This chapter introduces and explains important approaches and concepts that we use throughout subsequent chapters. These include program compilation, different execution modes, compiler optimizations, as well as heuristics used in compilers for different optimization tasks. Then, we describe dynamic linking and its implementation concept, as well as different languages, their intermediate representations and their runtimes. We will also explain GraalVM, the ecosystem on which our work is built.

2.1 Program Compilation and Execution

A compiler takes a program in a certain source language and translates it into a semantically equivalent representation in a target language. The target program consists of machine code, for example processor instructions for a specific target platform, or another intermediate representation (IR) such as Java bytecode [63, 113].

While machine code can be executed on the target hardware directly, programs in an IR or in a source language require an execution engine at run time. This can either be an interpreter, which simulates a machine for the given representation, or a just in time (JIT) compiler or dynamic compiler, which produces the machine instructions at run time when needed [63].

We will explain these different compilation techniques and their properties in the following subsections.

2.1.1 Ahead-of-Time Compilers

Ahead of time (AOT) compilers are also known as static compilers. Compiling code AOT means that the compilation happens before the code is executed. This is not only the oldest compilation approach, but has also been the most common approach for a long time [63]. The first popular compilers for high-level languages have been the Fortran compiler by John Backus [5] and Algol60, a compiler for Algol by Peter Naur [66].

Since then, lots of effort has been invested in AOT compilation research, resulting in compilers that are able to produce code with impressive performance [17, 33, 97]. This has also helped establish languages such as C and C++ that are among the most widely used programming languages [104].

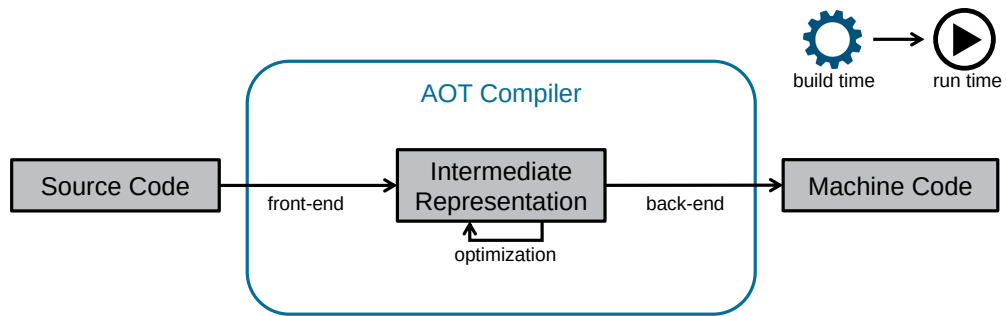


Figure 2.1: Basic structure of an AOT compiler: Source code is transformed into binary machine code. A common, but optional step is to use intermediate representations in between to enable optimizations.

The process of popular AOT compilers usually consists of different phases [63], as shown in Figure 2.1: The *front-end* splits the source language into tokens, parses and validates them, and creates an intermediate representation (IR), such as an abstract syntax tree (AST). In the next phase, this IR can then be analyzed and optimized. As a last step, the *back-end* translates the optimized program in IR and emits code specific to the target platform. In such a case, only the back-end has to be specific to the target platform, while the front-end and major parts of the optimization are platform independent.

Single-pass compilers perform a single traversal of the source code, directly emitting the target code. In contrast, multi-pass compilers process the code in multiple phase, often applying more sophisticated optimizations that require a global program analysis [113]. To support more complex optimizations and cross-platform code generation, multi-pass designs dominate modern compilers [63] such as clang [17] or gcc [33]. Nowadays, even languages that are usually JIT-compiled offer solid AOT compilers, such as Java [110] or C# [60].

2.1.2 Interpretation

Interpretation is an execution model in which source code or an intermediate representation (IR) is directly executed line-by-line by an interpreter, without prior compilation to machine code [63]. Examples for popular, real-world languages with an interpreter are Java [64], JavaScript [93] or Python [86].

Interpretation simplifies implementation, as it only executes code. As a consequence, the size of program artifacts is usually smaller than the corresponding compiled machine code. However, due to a per-instruction overhead and the lack of global optimization, interpretation is usually slower than compiled execution [93]. However, the flexibility of interpretation makes interpreted languages suitable in scripting or prototyping [109].

To achieve high peak performance, interpretation is often combined with JIT compilation, which will be covered in the next subsection. Nevertheless, performance of interpreters still has a high impact: Especially in continuous deployment, some application servers are restarted more than once per hour [27, 28]. Thus, runtimes that rely on interpretation in early iterations spend a substantial amount of time on interpretation.

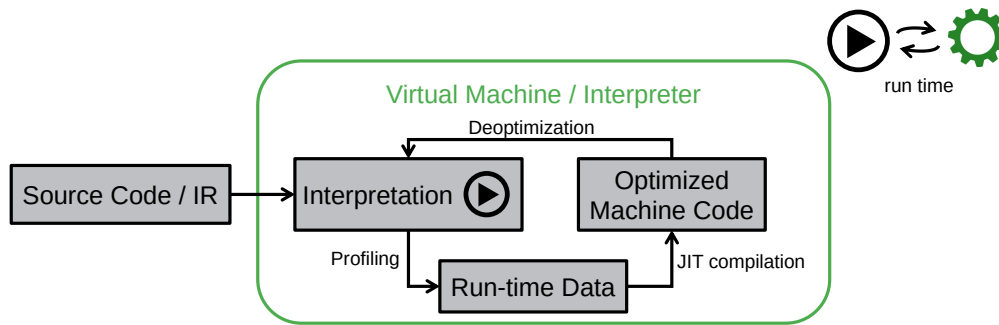


Figure 2.2: Basic procedure of a JIT compiler combined with an interpreter: The interpreter starts to execute the program. Profiling can detect often-executed code or paths within the code, and can compile them to optimized machine code. In case of a wrong assumption, code can be deoptimized, and execution reverts back to interpretation again.

2.1.3 Just-in-Time Compilation

Just in time (JIT) compilation, also known as dynamic compilation, happens at run time while the code is already executed by an interpreter [3]. The basic principle is shown in Figure 2.2. During interpretation, the runtime can collect information about the executed program, for example, which methods are executed often (*hot* methods), or which branches are taken. A JIT compiler can then compile frequently executed, *hot* code segments into optimized machine code at run time. As soon as JIT-compiled code is available, it replaces its interpretation, which leads to increased peak performance. It is even possible to replace actively executed code, which is called *on-stack replacement (OSR)*. However, run-time optimization comes with the drawback of a workload at run time, which usually leads to a worse warm-up performance compared to statically compiled code.

Modern JIT compilers often employ *speculative optimizations* to achieve an even higher performance [24, 26]. These optimizations generate code based on assumptions about the program's run-time behavior, such as frequently observed types or execution paths. For example, a JIT compiler might inline a polymorphic method call if a certain implementation is frequently executed, assuming its behavior remains consistent [26, 64].

However, these assumptions may not always hold true throughout the program's execution. When a speculative assumption is invalidated at run time, for example, when an unexpected type is encountered, the execution then falls back to a less optimized version of the code, which could be the original interpreted version or a less aggressively optimized compiled version [25]. This fallback is called *deoptimization* [43] and ensures correctness, as the less optimized code does not rely on the violated assumption. After deoptimization, the JIT compiler might re-evaluate the code and potentially re-optimize it later with new, more accurate runtime information, or with fewer speculative assumptions [29, 64].

Deoptimization is crucial for dynamic languages and environments where run-time behavior can be unpredictable. It allows JIT compilers to aggressively optimize for common cases, achieving significant performance gains, while still maintaining correctness when those common cases are not true [8, 29].

2.1.4 Properties of Execution Modes

Different execution modes have different advantages and disadvantages: Figure 2.3 shows AOT compilation, JIT compilation and interpretation as the corners of a triangle. Mixing two of these technologies usually leads to the advantage listed on the edge, while leading to the trade-off between the other two edges.

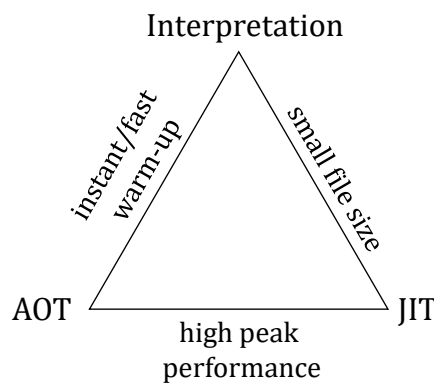


Figure 2.3: Three execution modes (AOT compilation, JIT compilation and interpretation) and their advantages, as in Figure 1.1.

Especially for Java applications, there is an effort to improve different execution modes in the Java Virtual Machine. As an example, Java HotSpot™ development is located on the right edge between interpretation and JIT compilation in Figure 2.3, tackling the trade-off between warm-up and peak performance. However, an AOT compiler for Java bytecode, GraalVM Native Image [110, 117], which is located at the bottom left in Figure 2.3, reduces warm-up times of Java applications by compiling ahead-of-time. However, as with most AOT compilers, the produced binary executables cannot be modified easily after their creation, and are often quite large compared to ordinary intermediate representations. Concretely, GraalVM Native Image does not support class loading, i.e., extending a running Java application with new types.

2.1.5 Native Extensions

A native extension is a module for a high-level programming language that is implemented in a compiled, low-level language (typically C, C++, or Rust) and loaded into the language runtime [38]. It exposes functions, classes, or types to the high-level language through the runtime's foreign function interface (FFI) or a dedicated extension API. The main reason besides getting access to certain native libraries is that performance-critical application parts can be written in optimized native code. In that way, they can combine interpretation and JIT compilation with AOT compilation and native execution.

2.2 Compiler Optimizations

Compiler optimizations are transformations that rewrite a program into a semantically equivalent form that is in a certain way better than its original representation [113]. The main goals are to reduce execution time, code size, and energy use, improve cache and branch behavior. The optimizations can happen at different levels: Source, IR and at machine code.

In this section, we briefly describe the most important optimizations that are used in modern systems and relevant for our approaches. Both optimizations presented below are no powerful optimizations if considered in isolation, but enable many powerful follow-up optimizations.

2.2.1 Inlining

Inlining is an optimization regarding method calls. When a method call is inlined, it is replaced by the code of the callee [6]. Actual parameters of the function call are assigned to the formal parameters of the inlined method and thus become local variables. Possible return values of the inlined method replace values returning from the former method call.

To perform inlining correctly, the implementation of the inlined method has to be known at optimization time, which is not always the case: In C++, Java or Python, for example, methods defined in a superclass may be overridden in a subclass. As a consequence, the dynamic type of the corresponding receiver object is not always known at compile time, since it can be of multiple classes. When such methods are virtual, i.e., the implementation is chosen by the dynamic type of the receiver object, also the corresponding implementation is not always known at compile time [23]. Thus, inlining is not possible in such a case.

Inlining removes method calls, which reduces stack and register manipulations due to arguments and method scopes. Moreover, integrating a method into another enables further local optimizations, as the optimizer analysis can now cover both method scopes at once [6].

However, inlining also has a disadvantage, which is the increase of code size. In the worst case, every method call gets inlined at multiple call sites, while the original callee is still kept as its own method. As a consequence of larger code size, the code itself has to be processed by subsequent optimizations, which requires more memory and compilation time. Therefore, many optimizers trace their inlining operations and stop inlining as soon as a certain limit, also called *inlining budget*, is reached [84]. Inlining is also one of the reasons why machine code is usually larger than its semantically equivalent IR. This motivates our approach for hybrid execution to reduce code size, which is covered in Chapter 3.

Table 2.1: Overview of existing compiler heuristics and corresponding important publications on which we build upon.

Approach	Publication
Static branch prediction	Wu and Larus [114]
Code size reduction by factoring out code	Debray et al. [22]
Optimization combinations affect code size	Foleiss et al. [30]
Automatic optimization selection	Stephenson et al. [95], Leather and Cummins [51]
ML for opt. level selection to reduce code size	Cugurovic et al. [19]

2.2.2 Profile-Guided Optimization

Profile-guided optimization (PGO), also known as feedback-directed optimization (FDO), is a compiler optimization technique that utilizes information about a program’s run-time behavior to make optimization decisions. By combining code analysis with profiling data, PGO can significantly enhance program performance and generate faster binaries [40, 54]. While being an essential part of JIT compilation, PGO is also integrated into static compilers such as GNU compiler collection (GCC), clang, and GraalVM Native Image as well [9, 31]).

The first step in PGO is to collect a profile of the program execution. This profile contains information about control flow, branch predictions, and memory access patterns. By knowing which parts of the code are most critical, the compiler can then guide a variety of optimizations: Functions that are frequently called from a specific call site can be inlined to eliminate the function call overhead [31], as described above in Section 2.2.1. Also, code can be laid out more effectively to benefit from caching effects [9, 94]. Moreover, PGO provides insights into branch probabilities [54].

2.3 Compiler Heuristics

The approaches in this thesis describe mixing execution modes to balance different trade-offs, for which a selection process between execution modes is necessary. Like many selection processes in compilers and runtimes, also our approaches are based on heuristics. Therefore, before proposing new heuristics, we revise and revisit existing heuristics that are relevant within our context. We are aware that many of the heuristics we inherit from the literature were crafted for different architectures, benchmarks and objectives. This makes published results often difficult to reconcile across papers. Still, we assume certain insights of existing research to be transferable and extendable to our hybrid execution approaches, and we will compare differences or unexpected outcomes.

An overview of existing compiler heuristics on which we build upon is given in Table 2.1. The following subsections discuss these approaches in more detail and highlight aspects which are relevant for our approaches.

2.3.1 Foundational Concepts

One of the first contributions in compiler heuristics may have been done by Backus et al. [5]. Since then, research has developed, and we will go through approaches relevant to ours in the following.

Lattner and Adve [50] contributed LLVM, which is another important foundational project for compilation. It is a flexible and widely-used infrastructure with a common IR (LLVM IR) that has become a standard for developing and implementing new compiler optimizations and heuristics. LLVM IR is low-level and platform-independent, and it is used by popular compilers internally such as clang [16, 17] or the Swift compiler [46]. We will describe LLVM in more detail in Section 2.5.1.

Our approaches in this thesis use LLVM as well, namely for hybrid execution of native extensions: When compiling C and C++ code to machine code, the clang compiler can also emit the intermediate LLVM IR during the compilation process. It is then possible to either execute the generated machine code, or interpret and JIT-compile the LLVM IR by a managed runtime. We describe how to mix these execution modes in Chapter 4, and we propose heuristics for an automated hybrid execution in Chapter 5.

Historically, static heuristics preceded the wide use of dynamic profiling. Foundational techniques for static profiling were introduced by Wu and Larus [114], on which many later approaches build upon. Dynamic profiling became popular afterwards [34]. However, static profiling remains valuable, since it does not require run-time instrumentation [55]. This relieves programmers from instrumenting and profiling their code dynamically. In addition, statically estimated profiling data is available at compile time, providing information that static compilers can exploit for optimization. For these reasons, static heuristics are important for our work.

2.3.2 Optimizations Along the Trade-off Between Size and Performance

Debray et al. [22] are one of the first authors proposing to use heuristics for code compaction. Aggressive interprocedural optimization and the abstraction of repeated code fragments led to a significant file size reduction of executables. They also present a flexible framework for identifying similar code fragments to be factored out, relying on heuristics to determine what constitutes an effective and beneficial abstraction. We are relying on the same idea as they do: Especially optimized AOT-compiled code results in large binaries, and we can reduce file size by not keeping the highly optimized binary code for all functions.

Foleiss et al. [30] also investigate how different combinations of compiler optimizations affect the final code size. They argue that evaluating optimizations in isolation is insufficient because their effectiveness is highly dependent on their interactions with other optimizations. The importance for our work in their approach is how they identify the complex relationships between optimization techniques to achieve a code size reduction.

2.3.3 Combining Different Optimization Levels

Code can be compiled and executed in different ways. There is also previous research on combining code resulting by compilation with different optimization levels, which is important for our work: The common aspect of our approaches in this thesis is to mix execution modes by combining different compilation results. Thus, we shortly describe approaches of combining different optimization levels in the following:

Stephenson et al. [95] use an evolutionary algorithm to search the complex space of possible heuristics. Their approach can create specialized heuristics for specific applications, but can also generate effective, general-purpose heuristics that improve performance on unrelated test sets. They inspire us to relieve compiler designers from the difficult and tedious task of manually tuning heuristics, which is the goal in Chapters 3 and 5.

Leather and Cummins [51] provide a retrospective on the use of machine learning in compiler optimization, making it highly relevant to the topic of compiler heuristics. They describe the evolution of the field from its beginnings, when search-based techniques were used to find optimal optimizations, to the development of machine learning models that could learn heuristics from those searches to achieve similar benefits quickly. The paper reviews key developments and looks toward the future of the field, providing a broad overview of how machine learning has been and will be used to automate and improve upon traditional, hand-crafted compiler heuristics.

While machine learning in compilers is often used for specific optimizations, Cugurovic et al. [19] apply machine learning to predict the hotness of a method. They let hot code be highly optimized, while cold code is compiled to a less-optimized version. Using the fact that higher optimization levels usually lead to larger executable binaries, they try to reduce the total code size while keeping performance as high as possible. Additionally, their implementation is based on GraalVM Native Image, an AOT compiler for Java bytecode. While this inspires our approach in Chapter 3, we even go a step further: Instead of selecting a lower optimization level for compilation, we do not even compile all code. By aiming to interpret cold code instead of compiling it, we also aim for a reduction in binary size while keeping performance, but try to also keep compilation effort low.

2.4 Linking and Lazy Function Resolution

Linking is the final phase of compilation, where parts of a program, such as object files and libraries, are combined into a single executable or shared library. Thus, it is necessary to resolve references between those single parts. The linker merges code and data sections, assigns memory addresses, and ensures that function calls and variable accesses point to the correct locations.

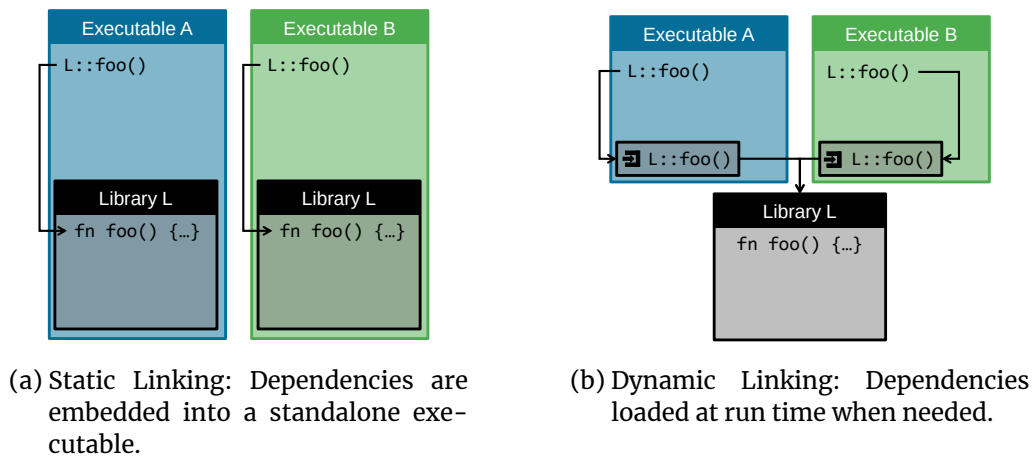


Figure 2.4: Static and Dynamic Linking in comparison.

This process can occur in two different ways: For static linking, dependencies are directly embedded into one binary and self-contained executable, as shown in Figure 2.4a. Thus, it happens before run time. In contrast, dynamic linking defers resolution to run time. On the first use of dependent code parts, the dynamic linker loads the corresponding dependent libraries into memory, resolves symbols, and patches references to their actual addresses. This means, functions are only looked up when first called. Dependent code is therefore not copied into every executable. Instead, it is sufficient to be present once per system, as shown in Figure 2.4b.

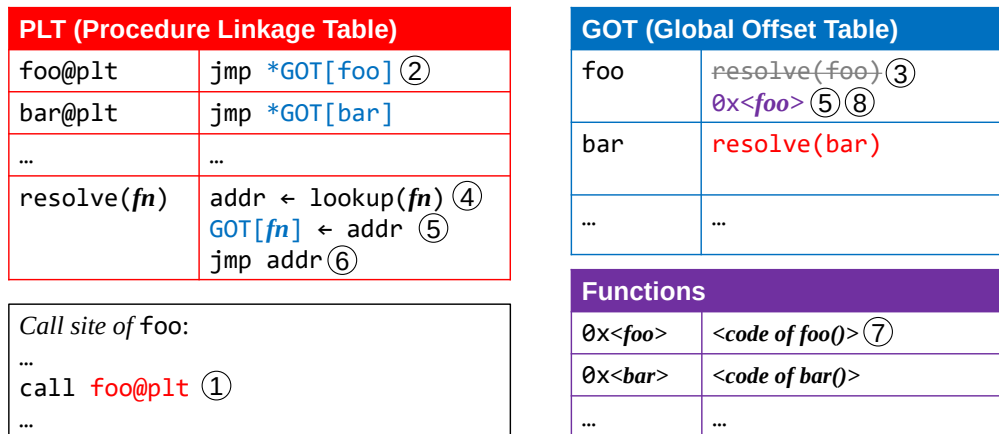


Figure 2.5: Dynamic function resolution using PLT and GOT.

Such a lazy resolution, however, requires additional data structures to be present, which will be explained in the following: The procedure linkage table (PLT) as part of the code section, and the global offset table (GOT) as part of the data section [1]. The way how they are used together will be shown in Figure 2.5 and explained in the following. Circled numbers within the explanation refer to the numbers in Figure 2.5.

For dynamic libraries, the implementations of our approaches use files in the format of executable and linkable format (ELF). In such ELF files, every call of a lazily resolved function is replaced by a call to the corresponding function entry in the procedure linkage table (PLT). Upon a call ①, execution continues in the PLT, where its corresponding entries ② perform a lookup into the global offset table (GOT). By default, every GOT entry points to the resolution stub in the PLT.

- If `foo` is called for the first time, the default GOT entry ③ triggers the resolution stub:
 - ④ First, the dynamic linker performs a lookup and fetches the actual address of `foo` in memory.
 - ⑤ Then, the corresponding GOT entry is overridden with the resolved address. This is possible as the GOT belongs to the data section, while the PLT and other functions are part of the read-only text section.
 - ⑥ As a last step, a jump to the resolved address leads to the execution of the actual function ⑦.
- If `foo` has been called before, the corresponding GOT entry ⑧ already contains the resolved address, such that the performed jump instruction in the PLT ② directly leads to the execution of the called function ⑦.

We can use these lazy linking to redirect calls at run time, which will be explained in more detail in Sections 3.5.2 and 4.5.3.

2.5 Selected Language Implementations

The implementations of our approaches described in Chapters 3 to 5 are based on systems for concrete languages or IRs. This subsection therefore covers language implementations that are relevant for our approaches, as well as their execution modes and engines.

2.5.1 LLVM

LLVM [50, 98] is a cross-platform compilation framework, built around its low-level IR LLVM IR. LLVM IR is used by various compiler front-ends and back-ends, and uses static typing and static single assignment (SSA) form [13, 21]. Usually, LLVM IR is compiled AOT.

Popular front-ends are e.g. clang [16, 17], which compiles C/C++ code to LLVM IR, or the Swift compiler [46]. LLVM IR also comes with a semantically equivalent, but more compact bytecode format, LLVM bytecode [98]. Our approach in Chapter 4 uses LLVM as a common and platform-independent IR, since there are existing systems for compiling and running LLVM bytecode in different execution modes [17, 91].

2.5.2 Python and its Implementations

Python [87, 107] has become a very popular programming language [104] and is not only, but as well used in machine learning [96]. Besides multiple Python language implementations, CPython [86], which is written in C, serves as the reference implementation.

While CPython prioritizes ease of use, its performance of the official Python benchmarks is mediocre [99], especially for computationally-intensive tasks. Another solution is to use the native extensions via the C API [89], allowing high-performance C or C++ code to be executed from Python programs. Furthermore, natively executed libraries such as NumPy [67] provide optimized numerical operations, further enhancing their performance.

JIT Compilation for Python Another possibility to improve the performance of Python is JIT compilation, as CPython offers since version 3.13 [88]. Also, GraalPy [42] or PyPy [85] offer JIT compilation with a copy-and-patch approach. On the official Python performance benchmark suite [99], GraalPy is 3.4x faster and PyPy is 4.8x faster than CPython [42]. GraalPy is the Python component of the GraalVM ecosystem and will be explained in more detail in Section 2.6.7. PyPy [85] uses meta-tracing, which can transform an interpreter into a tracing JIT compiler [12]. As is typical for tracing JIT compilers, linear sequences of operations in the interpreter are compiled. This concept can compile loops containing linear sequences of code into highly-efficient machine code. PyPy is written in a restricted subset of Python called RPython.

HPy Python's existing C API is optimized for CPython. If another Python implementation such as PyPy is used, the usage of the C API leads to a higher overhead. The HPy [45] project addresses this problem by introducing a new API layer that sits between the native extension and the Python runtime. Using HPy reduces the complexity of native extensions in Python.

Cython Cython [20] aims to blend Python and C: On the one hand, Cython syntax is similar to Python, aiming for easy usage. On the other hand, Cython can be compiled directly into C code, which leads to a high peak performance. Another common use case for Cython is to provide an interface for C/C++ libraries that are used in Python projects [20].

2.5.3 Java Virtual Machine

Java is a popular, statically typed and object-oriented programming language. Its standard toolkit, the Java development kit (JDK), includes the compiler (*javac*), the runtime including a garbage collector, and standard libraries. Java is used for server systems, Android development (via the JVM/ART and Kotlin/-Java), big data frameworks, and many enterprise applications [10, 105].

Java is compiled to Java bytecode that can be run on the Java virtual machine (JVM), which is the runtime for running Java bytecode on many platforms. The most widely used implementation is the Java HotSpot™ VM. It is the default JVM bundled with OpenJDK, the open-source reference for Java. It contains an interpreter, different garbage collectors [47], and two JIT compilers: The C1 compiler aims for fast startup and low compile time such that applications do not have to be run by the slower interpreter. The C2 compiler requires higher compile times, but performs highly intensive optimizations to achieve an eventually high peak performance instead [10, 105].

2.6 GraalVM

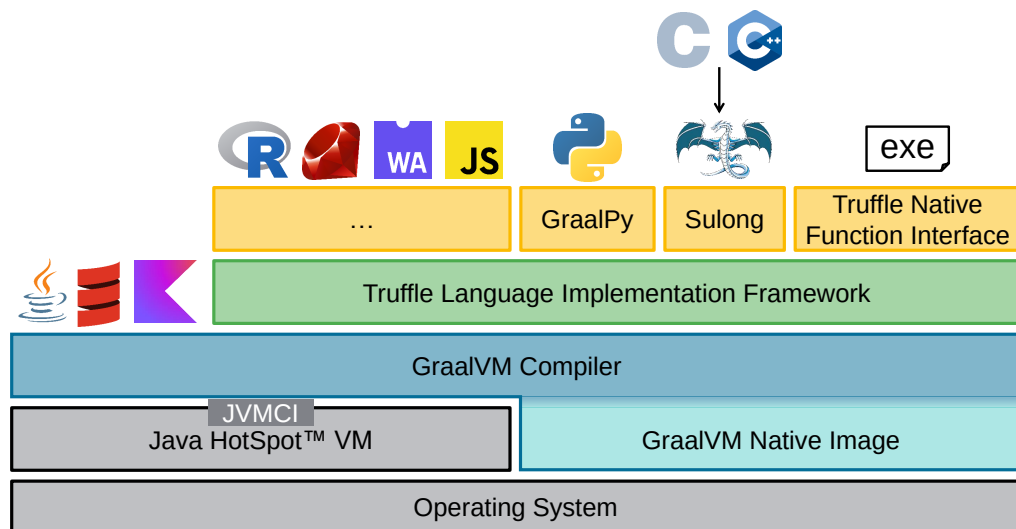


Figure 2.6: GraalVM ecosystem: Java bytecode can be used directly, whereas other languages require a runtime based on the Truffle Language Implementation Framework. The GraalVM compiler can serve as a JIT compiler for Java bytecode, while GraalVM Native Image compiles Java bytecode to machine code ahead of time.

GraalVM is a high-performance and multi-language ecosystem, containing different components shown in Figure 2.6. Its core is the GraalVM compiler, an aggressively optimizing JIT compiler for Java bytecode. On the one hand, the GraalVM compiler can be used as a JIT compiler when Java bytecode is run via the Java HotSpot™ VM. By using the Java virtual machine compiler interface (JVMCI), it can replace the C2 compiler. On the other hand, the Native Image component of the GraalVM compiler enables AOT compilation to a standalone binary executable as well.

As Java bytecode is the input representation for the GraalVM compiler, Java, Kotlin and Scala code can directly be processed. For other languages, the Truffle language implementation framework enables implementing an interpreter for custom languages that is optimized for the GraalVM compiler. Common runtimes are GraalJS for ECMAScript-compliant code, GraalPy for Python code, or Sulong as the engine for LLVM bitcode.

2.6.1 GraalVM Compiler

The GraalVM compiler is a JIT compiler for Java bytecode, written in Java [115]. It can serve as a replacement for the C2 compiler of the Java HotSpot™ VM via the Java virtual machine compiler interface (JVMCI). By using run-time profiling information for aggressive optimization, it can achieve a high peak performance: As a speculating compiler, it uses those run-time information to make assumptions about branching behavior, types or values of variables. This often allows subsequent optimizations, which further improves peak performance. In case an assumption turns out to be wrong, the compiled code is de-optimized, i.e., replaced by a less-optimized version, as described in Section 2.1.3.

The IR of the GraalVM compiler is called GraalIR [24]. It is based on a sea-of-nodes representation in SSA form [13, 18, 21], and is structured as a directed graph of data and control flow nodes.

2.6.2 Truffle Language Implementation Framework

Truffle [65, 112, 115] is a language implementation framework which enables developers to implement interpreters for programming languages [115], where usually an AST or bytecode is used as a source. It is based on the GraalVM compiler, thus the data structures produced by Truffle (and the corresponding language implementation) can be compiled directly by the GraalVM compiler. Languages for which Truffle implementations exist are for example Ruby, JavaScript or Python. The most important properties and tools of the Truffle framework are described in the following.

Optimization In the GraalVM compiler, dynamic optimization techniques are used for efficient compilation: Together with user code, the code of the Truffle interpreter for a certain language can be compiled using partial evaluation. This is the principle of first Futamura projection [32]. This enables compiling source code data structures such as an AST or bytecode to machine code.

Rapid Prototyping of New Languages Before new languages can be adopted, many things have to be manually implemented, including an optimizing compiler, debuggers, etc. Therefore, when implementations of approaches in this thesis are about to be extended to more languages, preexisting tools and the optimizing GraalVM compiler can be leveraged with little additional effort. Furthermore, if new developer tools (debug support, monitoring, ...) arise, they only have to be implemented once in Truffle and can then be used by all Truffle languages easily.

Polyglot Programming Different Truffle languages can interact with each other without performance disadvantages. This cross-language concept is explained in more detail in Section 2.6.4.

```

1 function parseTextTo2DPoint(inputstring) {
2     ...; return {x: xval, y: yval};
3 }

```

Listing 2.1: `pointParser.js` – JavaScript code that parses a string and returns a JS object.

```

1 struct Point { int x; int y; };
2
3 void mirror(struct Point *p) {
4     p->x = -p->x; p->y = -p->y;
5 };

```

Listing 2.2: `pointLib.c` – C code that modifies fields of a point object. The code is then compiled to LLVM IR/machine code (`pointLib.so`).

2.6.3 Sulong

Sulong can be used to execute code of native extensions, which is relevant for our approaches in Chapters 4 and 5. It is the implementation of the LLVM bit-code “language” on top of the Truffle framework, and thus serves as the LLVM runtime in the GraalVM architecture [91]. As it is based on Truffle, Sulong also supports features like high-performance compilation, debugging support, monitoring, and cross-language interoperability (cf. Section 2.6.4). By running on a JVM, Sulong also offers safe execution for unsafe languages such as C or C++ [91, 92]. Thus, errors can be observed more easily and automatically (as additional safety checks do not have to be inserted by certain tools or even manually).

As mentioned in Section 2.5.1, LLVM bitcode is an intermediate representation. Thus, Sulong is used to execute code of languages that can be compiled to LLVM bitcode, such as C/C++ (as shown in Figure 2.6) or Swift [75].

2.6.4 Truffle Interoperability

Truffle enables running code written in different source languages within a single runtime [37, 115]. Thus, it is possible to call functions across different languages. For example, cross-language function calls are shown in Listing 2.3, which contains the main method written in Python. Before invoking code from a foreign language, the code itself has to be loaded in lines 2 and 3. After that, a parser written in JavaScript is called, that parses text and returns a JavaScript 2D point object, as shown in Listing 2.1. This JavaScript object is then stored in a Python variable `p`. Then, in line 10 of Listing 2.3, the C function shown in Listing 2.2 is called with the JavaScript object `p` as a parameter. This C function is not called natively, but interpreted via Sulong and JIT-compiled by the GraalVM compiler.

```

1 #fetch references
2 jsFile = polyglot.evaluate(lang="js", src= "pointParser.js")
3 llvmFile = polyglot.evaluate(lang="llvm", src="pointLib.so")
4
5 #call a JS parser (returns JS object)
6 p = jsFile.parseTextTo2DPoint('{x:45,y:23}')
7 #p is now an object with fields x=45 and y=23
8
9 #call a C function from Python with JS obj
10 llvmFile.mirror(p)
11 #p.x is now -45

```

Listing 2.3: `main.py` – Example for Truffle interoperability: Python code that calls JavaScript code (Listing 2.1) to parse and C code (Listing 2.2) to modify the parsed data.

In contrast to other polyglot systems, Truffle language objects are not modified or marshalled when passed to a foreign language. This does not only keep the original semantics of the corresponding source language, but it also allows the dynamic compiler to perform optimizations across these boundaries, which contributes to high performance. In our example, the point object from JavaScript is neither converted to a Python object when it is stored in `p`, nor to a C struct when it is passed to the C/LLVM function.

Since Truffle uses the same IR for all language implementations, the code shown in the listings 2.1, 2.2 and 2.3 can also be JIT-compiled by the GraalVM compiler. This shows how different source languages can interact seamlessly, since cross-language function calls are compiled down to a single piece of machine code.

2.6.5 Truffle Native Function Interface

Truffle’s native function interface (NFI) [101] enables calling and executing AOT-compiled functions natively from a Truffle language runtime [36], as shown in Figure 2.6. Thus, the NFI can serve as a bridge between managed execution on GraalVM and native execution. It is mostly used for running code of existing native libraries, system APIs, or performance-critical code written in languages other than the Truffle language itself.

It is based on `libffi` [53], a FFI library. Internally, the NFI is implemented as another Truffle language that can be accessed via Truffle’s interoperability mechanism.

Key characteristics and functionalities of the Truffle NFI include [36]:

Efficiency and Flexibility It is designed to be faster and more flexible than traditional methods like the Java native interface (JNI). While JNI requires all interfaces to be known at compile time, the Truffle NFI enables calling functions without having to know the signature at run time [37]. The foreign function and memory (FFM) API [68] also enables calls without having to know the function signature at compile time, and performs calls faster than via the JNI.

Direct Native Code Invocation Truffle’s NFI enables direct calls to native code from foreign language applications, which is neither possible with the JNI nor the FFM API: Since the NFI is implemented as another Truffle language that can be accessed via Truffle’s interoperability mechanism, language can directly interact with native code without having to write boilerplate JNI code, FFM code, or similar bridging technologies [37].

Performance Optimization It creates call stubs for native functions that the GraalVM JIT compiler can optimize.

2.6.6 Sulong Toolchain

To conveniently work with Sulong, it has its own toolchain, which is based on clang [16, 17, 92], a C/C++ compiler front-end for LLVM. Thus, it first compiles C/C++ code to LLVM bytecode, which is then further compiled to machine code. The Sulong toolchain, however, also adds the LLVM bytecode as a special section to the generated machine code file. Thus, the resulting file contains both, LLVM bytecode and native code as shown in Figure 2.7. The presence of the LLVM bitcode, however, allows execution via Sulong, i.e. to interpret and/or JIT compile those functions.

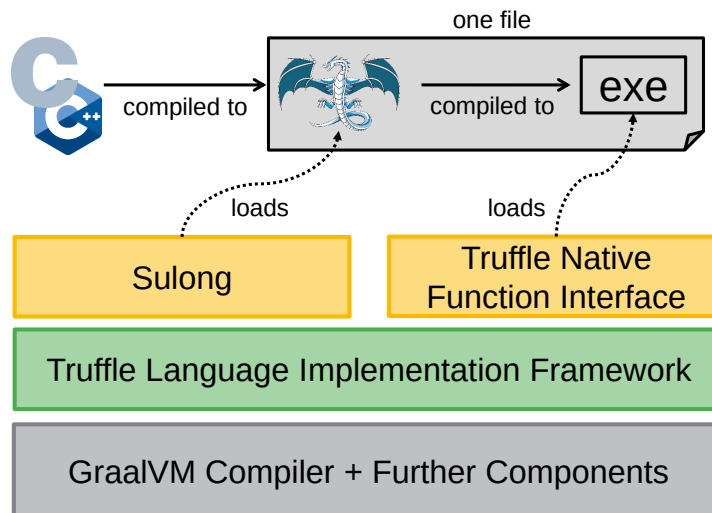


Figure 2.7: The Sulong Toolchain offers compilation of C/C++ code to machine code and includes the corresponding LLVM bitcode as well.

When such a file is loaded, Sulong executes LLVM bitcode by default. Only for functions that are not present as LLVM bitcode, the Truffle NFI is used to execute the corresponding native code.

2.6.7 GraalPy

GraalPy [42] is the Python language implementation on GraalVM, using the Truffle framework. The language runtime is based on Java, which is different from other popular Python implementations. As mentioned in Section 2.5.2, on average GraalPy outperforms the reference implementation CPython [42].

However, CPython achieves higher performance when it is combined with native extensions, especially with the NumPy [67] library. This is because most of the run time for NumPy is in the AOT-compiled native code, whereas the performance of the JIT-compiled Python code only has a minor influence in such applications.

Since C extensions are currently all executed via Sulong when LLVM bytecode is present, i.e., interpreted and JIT-compiled, we therefore propose a hybrid execution model for Sulong: Performance-relevant code parts of C extensions should also be executed as native code. We explain this concept in Chapter 4.

2.6.8 Native Image

GraalVM Native Image is a technology which compiles Java bytecode into a native executable. Instead of requiring a Java Virtual Machine (JVM) to interpret and compile code at run time, Native Image performs all compilation tasks ahead-of-time (AOT), creating standalone binaries that can run directly on the target platform [110].

Its instant warm-up capability enables applications to begin executing immediately, eliminating the traditional JVM warm-up period where code verification, class loading and interpretation typically occur. Warm-up times in a traditional JVM are usually two orders of magnitude longer than in Native Image [110, 117].

Particularly in cloud environments, immediate peak performance and resource efficiency are important criteria for software toolchains [27, 28, 73]. Thus, AOT compilation is also an important technology also for modern software development.

Creating a Native Executable

The process of creating a native executable from a Java bytecode application is shown in Figures 2.8 and 2.9 and works as follows:

GraalVM Native Image takes the code of an application in the form of Java bytecode. Certain aspects of a JVM are missing at run time with Native Image, such as a class loader. Thus, JDK code has to be provided as well. This is why not only application code, but also code of libraries, JDK and the SubstrateVM as shown in Figure 2.9 are fed into static analysis. SubstrateVM contains a minimal runtime for execution of the compiled program and will be further explained within this section.

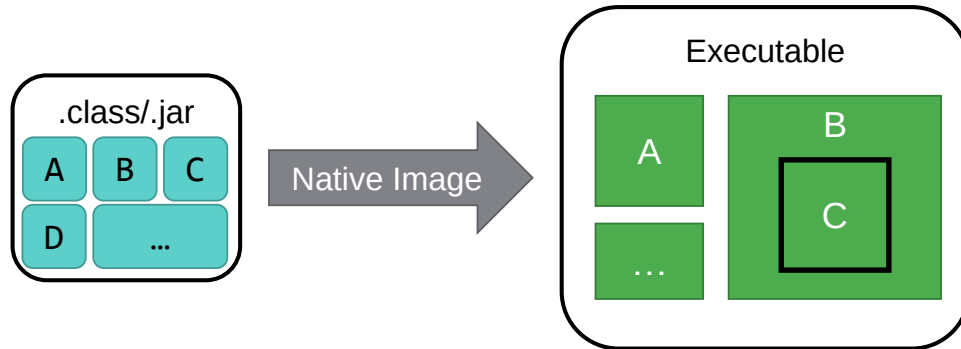


Figure 2.8: GraalVM Native Image compilation process from code perspective: Method A gets compiled regularly. Method C gets inlined into B. Method D could be proven to be dead code, so it is not included into the final executable.

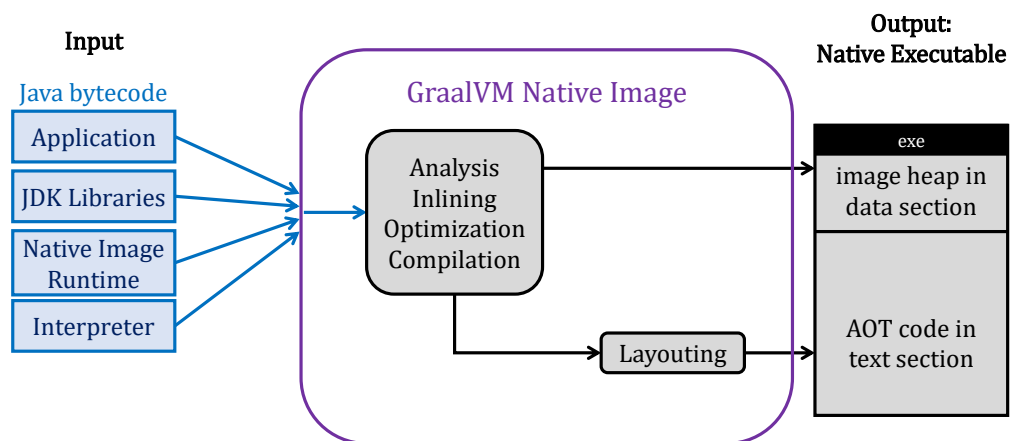


Figure 2.9: Native Image build process [117]: Java bytecode, including runtime code, is analyzed and optimized and then compiled to an executable.

Static analysis first performs a reachability analysis. These analysis steps are performed in a fix-point operation, i.e., they are applied repeatedly until the system stabilizes. Based on the result, further optimizations such as method inlining are applied. As an example, method C in Figure 2.8 gets inlined into method B. Since native image works on a per-method based compilation, every compiled method is called a *compilation unit*. In Figure 2.8, the compilation unit of B also contains code of method C. Once reachability analysis and optimization is done, initializations which can be done at build time are run, and a heap snapshot is taken.

After compiling those compilation units, they are laid out so that caching effects help with fast startup [9]. The heap snapshot after initialization is embedded into the data section of this file.

It is important to mention that GraalVM Native Image does not contain a class loader, so loading new types at run time is not possible. This is caused by the fact that the type hierarchy is already known during static analysis, enabling further optimizations. On the other hand, GraalVM Native Image has to guarantee that every reachable code on any branch has to be compiled and included at build time. For example, Java allows dynamically bound (i.e., virtual) method calls, which means that implementations of overridden methods often are reachable as well. Only if control flow analysis and type reachability prove that certain code can never be executed, that code can be excluded from the native executable. Due to trade-offs between precision and scalability during static analysis in Native Image, actually unreached methods can still end up in the final executable file [111]. Therefore, native executables produced by GraalVM Native Image are usually much larger than the individual class files that are loaded dynamically by the JVM.

SubstrateVM and its Debug Interpreter

SubstrateVM is a virtual machine (VM) written in Java, which contains the minimally required runtime components that are necessary in an executable produced by Native Image [72], such as a garbage collector.

Although SubstrateVM contains certain JVM components, it lacks mechanisms such as dynamic loading, reflection and deoptimization, making it simpler and faster. Since a binary executable is harder to debug than bytecode, there is the option to include the class files of an application into the executable, together with a lightweight bytecode interpreter that can then be used for debugging [69].

Profile-guided optimization (PGO) within GraalVM Native Image

PGO has been described above in Section 2.2.2. Like other static compilers such as GCC or clang, GraalVM Native Image uses PGO to optimize the produced executable¹.

¹To use PGO with GraalVM Native Image, the enterprise version is necessary. PGO is not available for the community edition [70].

The PGO process involves multiple stages [110], starting with a special build with profiling counters. This special build is then executed with representative workloads. During these *agent runs* the profiling counters collect data on the behavior of the application. The agent records, for example, which classes are loaded, which methods are invoked, which reflection tasks are performed, or which files are accessed. More importantly, the profiler can collect control flow information to improve branch prediction. Upon completion, these runs generate a file in JSON format containing the collected profiling information. Afterwards, a subsequent optimized build uses this generated profile data to guide optimizations, such as inlining of frequently executed method calls or removing code proven not to be used, to enhance application performance.

2.7 Statistics

In the following, we describe statistical concepts that will be used in the following chapters.

2.7.1 Box Plots

A box plot is a graphical way to show the distribution of numerical data. It is based on dividing the data into quartiles: The box extends from the first to the third quartile (i.e., the interquartile range) and thus represents the middle 50% of the data. The line in the box denotes the median. The whiskers extend to the farthest data point within $1.5 \times$ the inter-quartile range from the box. Circles denote data points outside of the whiskers. Figure 2.10 shows sample data values $\{1, 3.5, 5, 5.7, 6, 6.6, 7, 8, 9\}$ and their corresponding box plot.

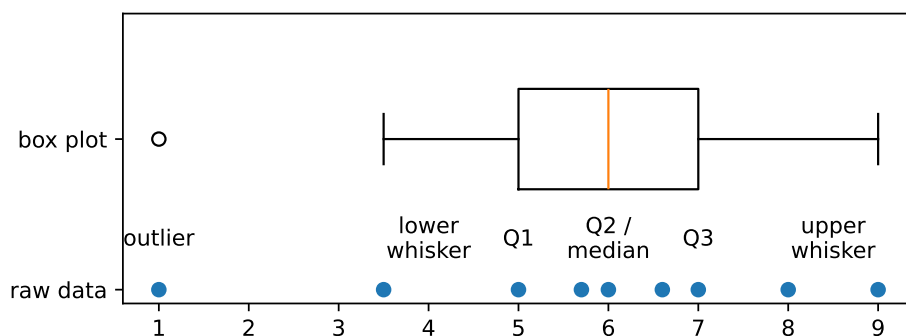


Figure 2.10: Sample data values and their box plot with box, whiskers and one outlier.

2.7.2 Correlation Coefficients

To see whether two features correlate, which is important when we, for example, compare method features, one can use correlation coefficients. Pearson's and Spearman's correlation coefficients both provide a unitless number: +1 denotes a perfect positive linear correlation, -1 a perfect inverse, linear correlation. 0 indicates that there is no correlation. While Pearson's coefficient indicates linear correlation, Spearman's coefficient works on ranked data. To calculate Spearman's coefficient, the data has to be sorted, and data processing is not continued with the raw data as such, but with the rank of every value within the feature. Both coefficients are computed as shown in Equations (2.1) and (2.2), respectively.

$$\rho_{\text{Pearson}}(X, Y) = \frac{\text{cov}(X, Y)}{\sigma_X \cdot \sigma_Y} \quad (2.1)$$

$$\rho_{\text{Spearman}}(X, Y) = \rho_{\text{Pearson}}(R[X], R[Y]) \quad (2.2)$$

2.8 Convention of the Terms Runtime, Run Time and Run-Time

The terms *runtime*, *run time*, and *run-time* are closely related, but are used with slightly different meanings throughout this thesis.

- The closed compound *runtime* is used as a noun referring to the execution environment. For example, the Java runtime provides memory management.
- The open compound *run time* is used as a noun referring specifically to the duration of a program's execution or a measured execution time. For example, the run time of an application is 2 seconds, and profiling data is collected at run time (i.e., dynamically).
- The hyphenated form *run-time* is used as an adjective modifying another noun. For example, we can perform run-time optimizations, i.e. optimizations at run time.

This convention is adopted consistently throughout the thesis for clarity and readability.

Chapter 3

Mixing Execution Modes to Reduce Binary Size

Our thesis is about mixing execution modes to balance different trade-offs. In this chapter, we look at the trade-off between size and performance. Thus, this chapter describes a hybrid execution approach for Java programs to reduce the size of binary executables while keeping peak performance high. We tackle this problem by storing less-frequently executed code in more compact representations. Performance-critical code is fully compiled to native code and thus still executed in a highly-optimized version.

The goal is to find heuristics to automate the decision whether to compile or not to compile certain code. This decision should reduce the binary size while keeping peak performance high. This chapter looks at both, static as well as dynamic heuristics and their evaluation.

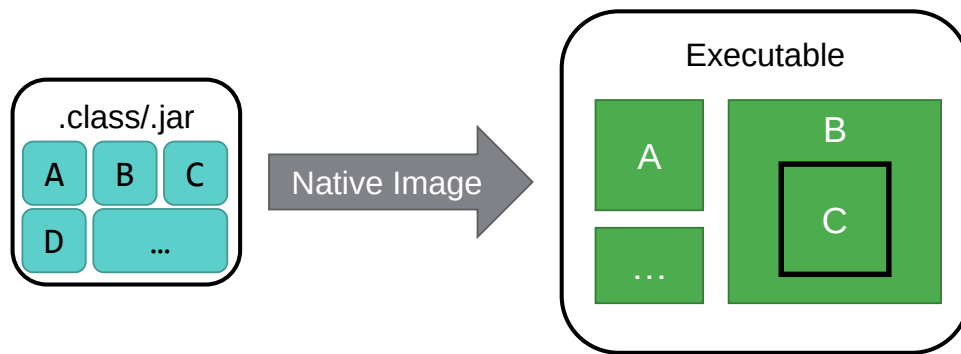
3.1 Motivation

As mentioned in Sections 1.1, 2.1.1 and 2.6.8, AOT compilation can produce highly-optimized binary executables that achieve a high peak performance, while start-up effort is reduced to a minimum. However, the resulting binary files can get quite large: Every code gets optimized on a high level, even code that is never called or only executed rarely. Shrinking the size of AOT-compiled executables without sacrificing performance is an ongoing research task [19, 22, 81], which is also the goal of this chapter.

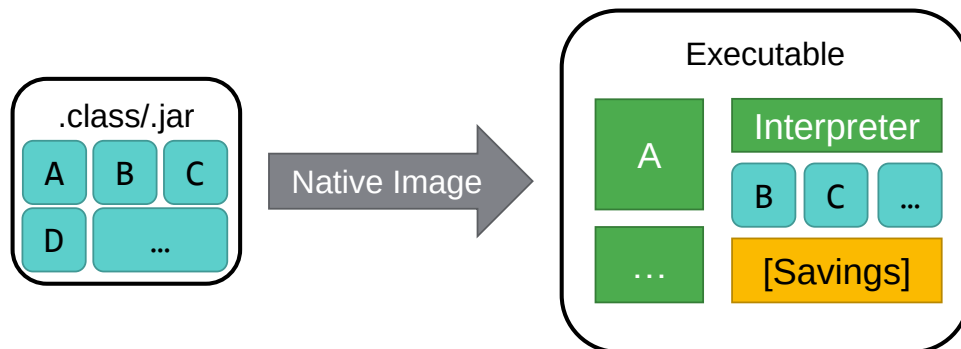
Our approach aims for smaller native executable files of Java programs by replacing compiled methods. However, the calculation is not as simple, since interpretation of the removed code requires it to be present as Java bytecode. The total file size will only drop if the Java bytecode is smaller than its compiled machine code. To validate that, the code of the DaCapo benchmark suite [11] is analyzed in the following by comparing the corresponding code sizes per compilation unit.

The result can be seen in the scatter plot shown in Figure 3.2. While the x axis shows the size of Java bytecode, the y axis denotes the size of the compiled code in bytes. The compiled code of the majority of the methods is roughly 3.1 times as large as the corresponding bytecode.

The idea to tackle this problem is to include a lightweight interpreter into the native executable, as shown in Figure 3.1: The standard compilation process is



(a) Regular GraalVM Native Image compilation process as in Figure 2.8.



(b) Compilation process when replacing compiled code with bytecode.

Figure 3.1: Compilation process of GraalVM Native Image, with and without our approach.

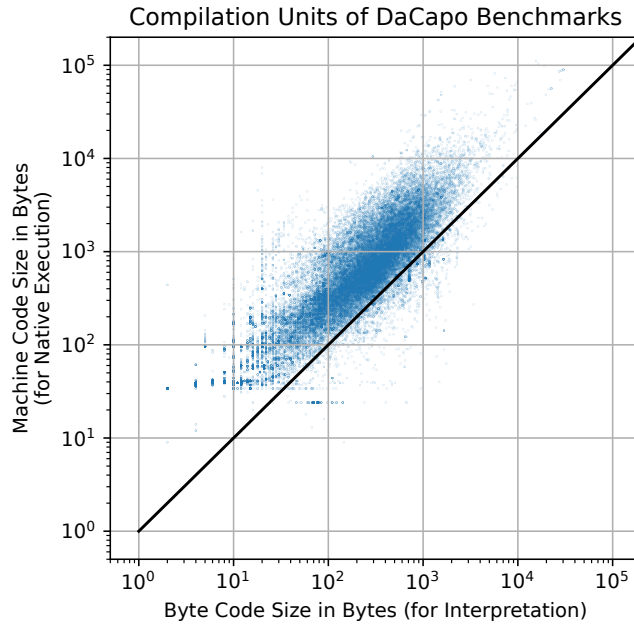


Figure 3.2: Comparing code sizes: bytecode vs. machine code. On average, machine code of a compilation unit is larger than its bytecode.

Table 3.1: Basic principle of hybrid execution.

	Hot Code	Cold Code
Execution Mode	AOT Compilation	Interpretation
Runtime Environment	Native Executable	Runtime Interpreter
Results in	High Peak Performance	Reduced Binary Size

shown in Figure 3.1a and has been described in Section 2.6.8. Our approach is to replace some of the compilation units by their smaller bytecode, as shown in Figure 3.1b. To execute the bytecode, an interpreter is needed as well. Thereby, not all code has to be included as native code in the executable, but some code can be stored in a less-optimized and more compact intermediate representation. Our goal is to reduce the size of the executable, while keeping peak performance reasonably high. This is possible if less-frequently executed code is interpreted, while performance-critical code is compiled AOT, as also shown in Table 3.1.

The resulting executable can then be tuned between performance and image size. In other words, if performance is critical, all code is AOT-compiled. If storage is rather limited, less frequently executed code can be interpreted. If the decision to interpret certain code turned out to be wrong, code can be compiled again. Our approach does not impair execution correctness, since only execution modes are modified, while execution semantics do not change.

While it is better to aim for size reduction in memory-critical systems, there are also scenarios where peak performance has a higher priority than resource efficiency. Thus, the result of our approach does not only depend on the code, but also on the context of its execution.

3.2 Code Features and Correlations

As a first step, we analyze run-time metrics and static code features of compilation units to identify promising correlations from which we can derive heuristics in Section 3.4. The goal of the heuristics is to identify compilation units that can be interpreted to shrink the executable size while avoiding a major impact on run-time performance.

We will first introduce the investigated code features and run-time data we use in the analysis, and then discuss the correlations we found.

3.2.1 Methodology

In the analysis, we focus for most parts on the largest DaCapo [11] benchmark *fop*. We chose *fop* since it has the most compilation units from the DaCapo benchmarks, and thus, provides the best opportunity to identify correlations. While focusing on a single benchmark may prevent us from seeing other useful correlations, it also reduces the risk of overfitting heuristics to the overall benchmark set. Since the DaCapo benchmarks originate from different code bases, they are likely to differ in their various characteristics from each other.

We extract the data for our analyses during the build process of a GraalVM Native Image executable. As shown in Figure 2.9, the Native Image compiler first performs global analyses, for instance to determine code reachability with a closed-world assumption. Then, the code is grouped into compilation units, each unit consisting of a node graph of the Graal IR. These units are then compiled into relocatable machine code. At this point, we collect the static features. After that, only those compilation units are laid out and included into the final executable where our heuristics (described in Section 3.4) decides them to be used instead of interpreted.

While conceiving the analyses that we describe below, we ended up collecting the following features. They are based on individual compilation units, i.e., method and all its inlined methods, and rely only on static information. Thus, they are also candidate inputs for the heuristics that we will derive from our analyses later on in Section 3.4.

- `nCycles`: number of estimated CPU cycles, estimated by a static heuristic [52, 61].
- `compiledSize`: size of the compiled code (in bytes)
- `nPars`: number of parameters
- `name`: method name
- `minRetDist`: shortest node distance from entry point to a return node
- `maxRetDist`: longest node distance from entry point to a return node
- `nNullCmp`: number of nodes that compare a pointer against null

3.2.2 Run-Time Data: Execution Time Ratio

Since we want to avoid a major impact on run-time performance, we look at the difference in execution time a compilation unit takes when it is executed on the

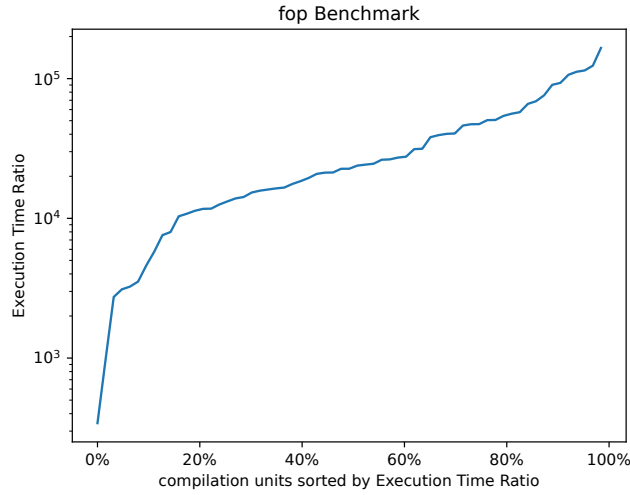


Figure 3.3: Distribution of the ratio $\frac{\text{interpreter execution time}}{\text{AOT execution time}}$ for compilation units of the DaCapo *fop* benchmark. It includes all compilation units activated at least once. It shows that for more than 90%, the benefit of AOT compilation is larger than a factor 400.

interpreter instead in its AOT-compiled form directly on the CPU. Specifically, we assess the benefit of compiling a compilation unit by determining the ratio between interpreted and AOT-compiled execution time, $\frac{\text{interpreted time}}{\text{AOT execution time}}$. The AOT execution time is measured with `perf` [100], as functions of a native executable are measured. To track the execution time of interpreted compilation units, `java.lang.System::nanoTime` is used.

In both cases (AOT-compiled and interpreted code), only the self time of a compilation unit is measured, i.e. time of callees of a compilation unit is not included. Compilation units without activations are excluded from this analysis. As a drawback, only compilation units with at least one call are captured (since $\frac{0}{0}$ is not a number).

From a performance perspective, units with larger execution time ratios yield greater benefit from compilation and should be prioritized. The distribution of those ratios for the DaCapo [11] *fop* benchmark is shown in Figure 3.3.

This execution time ratio is rather high, reaching from $400\times$ up to $10000\times$, with a median of $2000\times$. One reason for that is that we used a lightweight, research prototype interpreter that does not perform any optimizations.

In contrast to the static features listed above, the execution time ratio cannot be computed at build time, since it requires us to measure the run time. Thus, we use it only to identify correlations, but not in the definition of heuristics.

3.2.3 Correlations of Static Features and Execution Time Ratio

For the following sections, we analyze how static features related to the execution ratio of 3.2.2. To illustrate correlations, we include scatter plots, where every dot represents a compilation unit. While the x axis represents the static

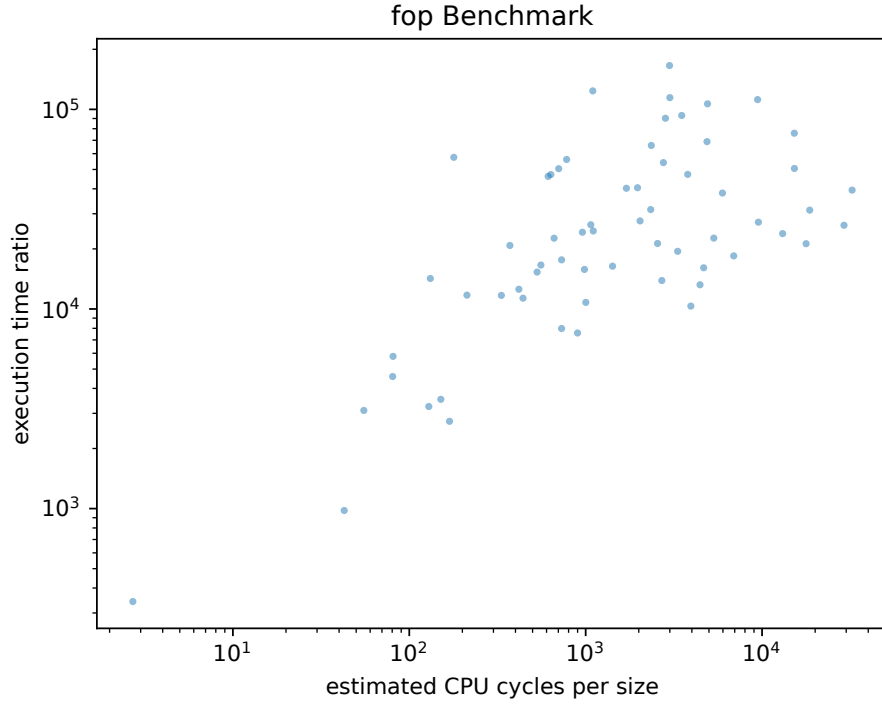


Figure 3.4: Estimated computational density of cycles per byte compared to the execution time ratio. Higher density, i.e., more estimated CPU cycles per size benefits more from AOT compilation.

feature, the y axis indicates the execution time ratio. We also provide Pearson’s and Spearman’s correlation coefficients, which have been described in Section 2.7.2. Spearman’s correlation works on ranked data.

Estimated CPU Cycles per Byte

As a first property, we investigate how computationally dense a compilation unit is. The idea is to see whether there are compilation units that are large but do not perform a proportional amount of computation, and thus, would be candidates for interpretation.

For our purposes, we only look at the size of a compilation unit in bytes and the cycles the Graal compiler estimates for it based on its built-in heuristics. To see whether interpretation could make sense, we plot $\frac{nCycles}{compiledSize}$ against the execution time ratio in 3.4. As expected, the plot indicates that more computationally dense compilation units cause a higher overhead when they are interpreted. The correlation coefficients are +0.16 (Pearson) and +0.64 (Spearman). This motivates that we found a feature with a strong possible correlation.

Number of Parameters

A feature that the machine-learning-based approach of Cugurovic et al. [19] identified as potentially relevant is the number of parameters of a compila-

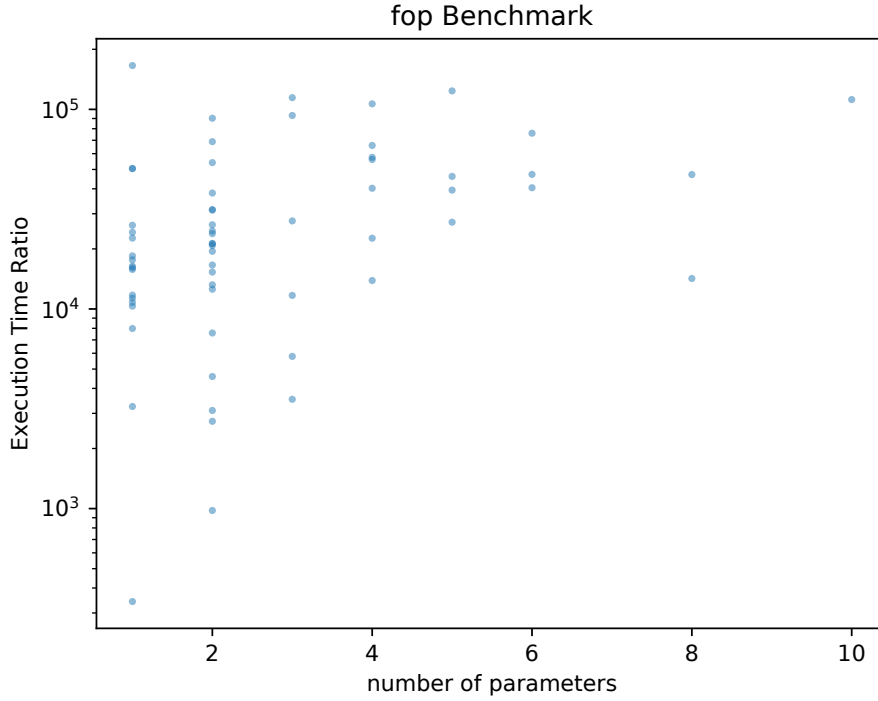


Figure 3.5: The number of parameters of a compilation unit compared to the execution time ratio. The more parameters a compilation unit has, the more compilation pays off.

tion unit. One may speculate that there is a deeper correlation between number of parameters and complexity of a method. Figure 3.5 shows the execution time ratio versus the number of parameters. The scatter plot shows a somewhat triangle-like distribution. Since the number of parameters is a dimension with only low resolution, it appears almost line-like in the plot and suggests that a low number of parameters can have a low and high execution time ratio. However, it also shows that methods with higher number of parameters typically benefit strongly from compilation. This is supported by the correlation coefficients $+0.38$ (Pearson) and $+0.41$ (Spearman).

Early Return

When analyzing the execution time of compilation units, we noticed that the first activation of a compilation unit can take much longer than subsequent ones, as shown in Figure 3.6. Every line represents a compilation unit, the x value denotes the number of the activation. The y value is the relative duration of a single activation w.r.t. subsequent activations; it is computed as in Equation (3.1) ($\text{time}(n)$ denotes the time of the n^{th} activation, N is the total number of activations).

$$\text{relative time of } n^{\text{th}} \text{ activation} = \frac{\text{time}(n)}{\frac{1}{N-1} \cdot \sum_{i=2}^N \text{time}(i)} \quad (3.1)$$

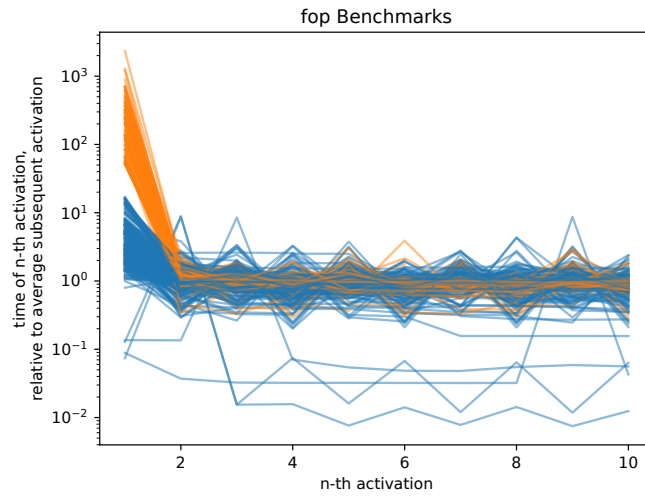


Figure 3.6: Activation Times of Compilation Units: There are some compilation units (marked orange) whose first activation takes much longer than subsequent ones. The y value is computed as in Equation 3.1.

```

1  static boolean initialized = false;
2  void some_init_method() {
3      if(initialized) {
4          return;
5      }
6      //...do intensive init work
7      initialized = true;
8  }

```

Listing 3.1: Lazy Initialization Pattern in Java. A presumed candidate pattern for detection.

While a first execution usually takes more time for initialization, there is also a group of compilation units, marked orange in Figure 3.6, where their first activation takes much longer than for other compilation units (marked blue).

Seeing the first activation overhead, we hypothesized that these compilation units perform some form of initialization during the first activation. Perhaps as sketched in Listing 3.1.

To identify code patterns with early returns more generally, we count how many nodes it takes to connect the compilation unit's entry point with all its returns and compare the shortest to the longest distance. Based on this, we calculate an early return ratio $\frac{\text{minRetDist}}{\text{maxRetDist}}$. A value close to zero indicates an early return, while a value close to one indicates that all paths to a return are equally long.

The scatter plot in 3.7 compares the early return ratio to the execution time ratio. Surprisingly to us, it shows an inverse correlation with correlation coefficients of -0.33 (both, Pearson and Spearman). Thus, it is the opposite of what we initially expected. This suggests that even if there is an early return, the code is often computationally intensive.

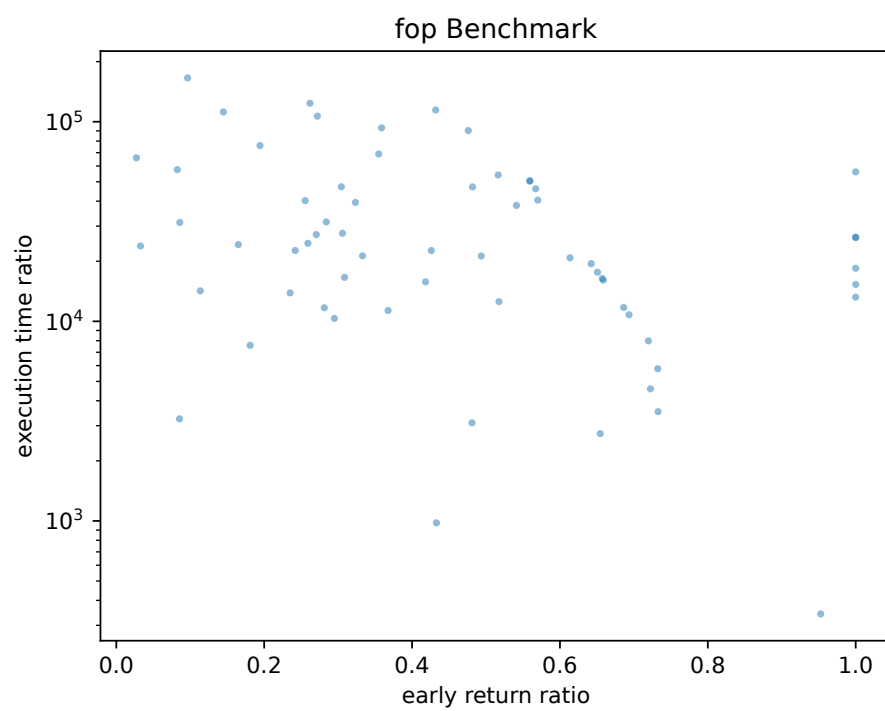


Figure 3.7: Scatter plot showing the early return ratio compare to the execution time ratio. The data suggests that the earlier a return can be reached, the more compilation pays off.

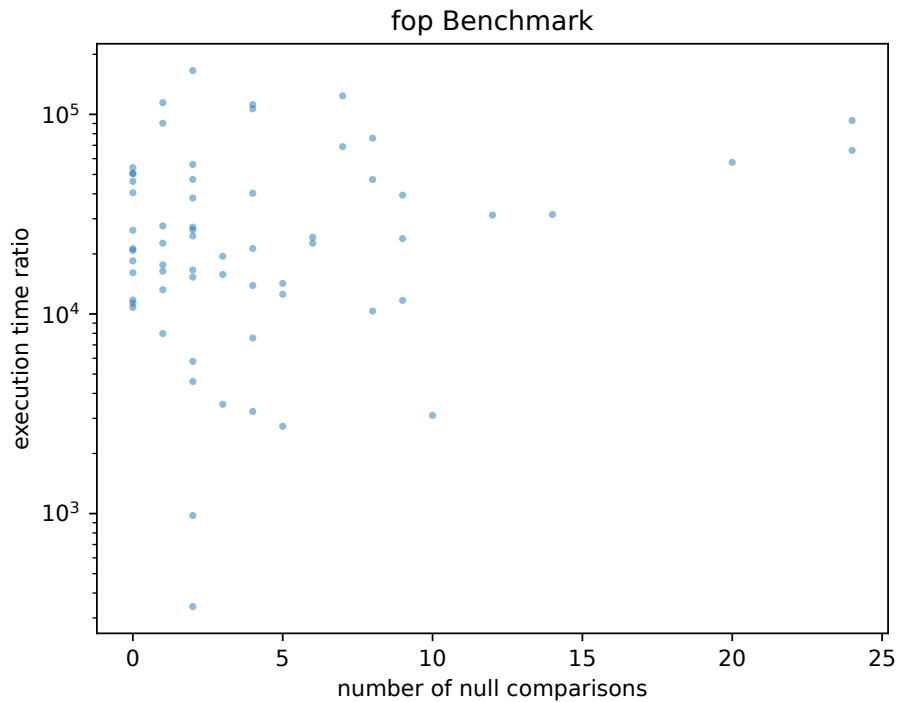


Figure 3.8: Scatter plot showing the number of null checks compare to the execution time ration. More null seem to indicate a higher benefit of compilation for compilation unit.

Pointer Null Checks

Inspired by the early returns, we also investigated pointer comparisons with null, which are also used in literature [14, 114]. The idea here is that a null check may indicates a cold branch, as null pointers may not be expected and lead to the rare error handling case.

There is only a rough correlation in the scatter plot in Figure 3.8, the coefficients are +0.21 (Pearson) and +0.07 (Spearman). Thus, a higher number of null checks only slightly correlates with a higher execution time ratio.

3.2.4 Static Initializers

Static initializers in Java are methods for class-level data initialization. For most static initializers, our measurements did not capture an execution time ratio, since they were not executed. This already suggests that they could be candidates for interpretation.

Generally, static initializers in Java combine the code of static field initializations and `static{}` blocks. The Java-to-bytecode compiler collects and merges all such initialization code into a single method, referred to as `<clinit>` in the bytecode. This method is invoked lazily when needed, thus, at most once. If a class is never referenced at run time, its static initializer is never executed, too.

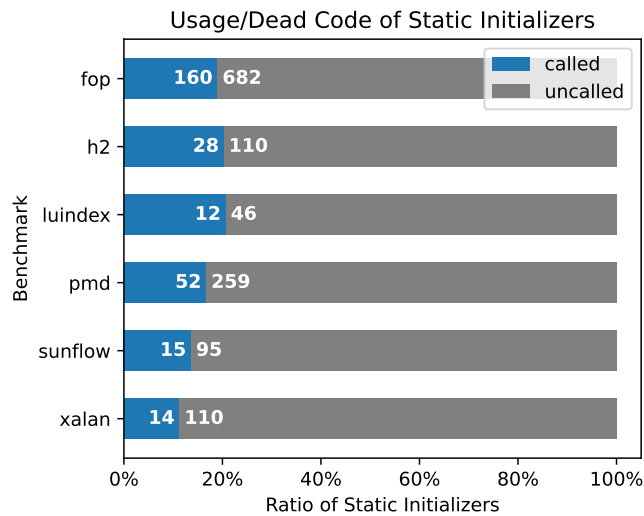


Figure 3.9: Usages of static initializers: In the DaCapo benchmarks, only 12% to 20% of the static initializers are actually called.

Also for GraalVM native image, `<clinit>` compilation units (i.e., static initializers and the hull of their inlined methods) can be invoked lazily. We see several reasons why it may be beneficial to avoid compiling static initializers:

At most one activation

As explained, static initializers get executed once at most. Thus, they can easily be considered as cold code in general.

High ratio of unused code

If a class is never referenced at run time, its static initializer remains unused during the current execution.

For the DaCapo [11] benchmark suite, Figure 3.9 shows the ratio of executed vs. unused static initializers. It shows that only $\approx 10 - 20\%$ of the static initializers get called. There are two reasons for such a low usage. On the one hand, Native Image uses a rather conservative cut-off in its reachability analysis [111] as mentioned in Section 2.6.8. Thus, some code that might be considered dead still remains in the image, because the analysis cannot prove it.

On the other hand, some Java classes are only used for certain states in the application, e.g. an error case. If the application never reaches such a state, but could reach it with different input parameters, then the static initializer often remains unused as well, although it cannot be considered to be dead code.

Large Code Size

As static initializers often initialize static fields with either constant values or values without high computing effort, machine code can get rather large. As an

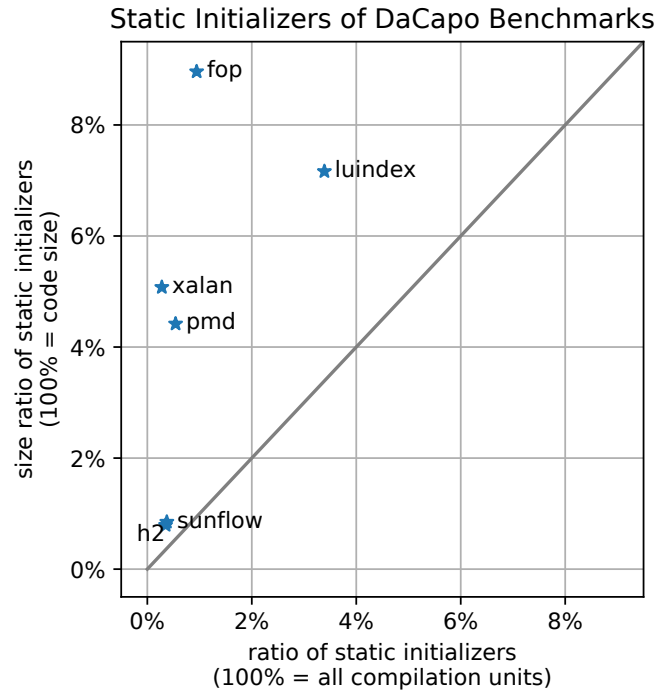


Figure 3.10: Sizes of Static Initializers in DaCapo Benchmarks: On average, static initializers are larger than an average method.

example, Figure 3.10 shows the size property of static initializers for different benchmarks of the DaCapo suite. The x axis shows how many compilation units are static initializers within the application, the y axis shows their combined code size normalized to the whole application.

For the *fop* benchmark, there are 842 static initializers out of 88955 compilation units. However, those $\approx 0.94\%$ compilation units take almost 9% of the overall image size.

The two largest static initializers `org.apache.batik.xml.XMLCharacters` and `sun.awt.X11.XKeysym` are responsible for 2% of the image size, even though they are never called when our benchmarks are executed.

Thus, since static initializers are rarely executed, but large compilation units that are executed at most once, they seem to make good candidates for interpretation.

3.3 Research Questions

We replace executable compiled code by interpretable bytecode to shrink the sizes of executables. With regard to that, we are interested in the following aspects:

1. What is the trade-off between size and performance when applying static heuristics for hybrid execution based on the code features of Section 3.2?

2. What is the largest possible size reduction of applications without significantly impacting run-time performance? Knowing the maximal reduction also allows us to ask the following:
3. How close do static heuristics approach a close-to-optimal solution, which is based on dynamic information?

3.4 Heuristics

Focusing on research question 1 and based on the analyses in Section 3.2, we formulate heuristics to reduce the binary size and enable us to decide, whether to compile or not to compile a compilation unit, while keeping peak performance high. As described already in Section 3.2.1, the decision for static heuristics happens after the code has been compiled and before it has been laid out and stored into the final executable. The reason for that is that the full information is already available (compiled size). In future, we could try to estimate necessary information such as the compiled size of a compilation unit at an earlier compilation stage. For dynamic heuristics, the call frequency of every compilation unit is collected from dynamic information of profiling runs done with instrumentation.

3.4.1 Static Decision: Per-Compilation-Unit Compile Score

Our approach for static heuristics is to estimate a compilation-worthiness order of all compilation units at build time. Since the initial global analysis gives us a list of all compilation units, this allows us to rank them for the subsequent decision of whether to compile or not.

Thus, our heuristics need to map every compilation unit *cu* to a corresponding score between 0 and ∞ . When a compilation unit is considered to be more important to be compiled by a heuristic, its score will be higher. A value of 0 indicates that the compilation unit should not be compiled. On the other hand, some compilation units of the runtime system have to be compiled, which is indicated by a value of ∞ . This is true for some compilation units of the runtime system that are not supported to be interpreted, such as the interpreter itself (as described in Section 3.5.3) and core parts of the Native Image runtime [110].

We can then sort the compilation units by this score, and choose compilation units with the lowest score first to be interpreted instead of compiled. As size information (*baseSize*, as well as the size of each compilation unit) is already available at build time (see Section 3.2.1), we know how many units to interpret to achieve the intended size reduction. The *baseSize* is the size of the overall image, which includes the native code as well as objects that were created as part of the initialization during the compilation, i.e., the heap snapshot (see Section 2.6.8).

We evaluate the effects on peak performance for given heuristics and size reduction *sizeRed* $\in [0\%, 100\%]$ by using the following algorithm to select compilation units for interpretation:

1. Receive the list of compilation units during AOT compilation, as described in Section 3.2.1.

2. Sort the list by the chosen heuristic score function.
3. Set `goalSize` to `baseSize · (1 − sizeRed)`.
4. Set `curSize` to `baseSize`.
5. While `curSize > goalSize`:
 - a) Take the next candidate for interpretation, i.e., the one with the lowest score in the list.
 - b) Set the execution mode of compilation unit to interpretation.
 - c) Update `curSize` by replacing the machine code size with the bytecode size.
6. Set all compilation units without an assigned execution mode to AOT compilation.

With this approach, we obtain an executable with the desired size reduction. The peak performance can then be compared to the baseline, i.e., the same application where all compilation units are AOT-compiled.

3.4.2 Single-Feature Static Heuristics

Based on the analysis in Section 3.2, we create five heuristics. As a kind of control group, we will also include a *random* heuristic, which we also briefly explain here. A summary of these single-feature heuristics is given in Table 3.2.

Estimated CPU Cycles per Byte: `ecyc` (based on Section 3.2.3) The positive correlation we saw indicates to us that more computationally dense compilation units cause a higher overhead when they are interpreted.

To smoothen the heuristic and account for the orders of magnitude difference in computational density, we take the logarithm of the ratio. Thus, the score value for this heuristic is determined by $\log\left(\frac{nCycles}{compiledSize}\right)$.

Number of Parameters: `pars` (based on Section 3.2.3) While the correlation is not as clear as for `ecyc`, a higher number of parameters for a compilation unit seems to indicate a high interpretation overhead. To avoid a value of zero, the score value for this heuristic is determined by the number of parameters as such, incremented by one (`nPars + 1`).

Early Return Ratio: `rt rn` (based on Section 3.2.3) For the early returns, there is a negative correlation between the early return ratio and execution time ratio. Therefore, we use a heuristic that leads to compilation units with early returns to be more likely to be compiled. Specifically, we use the inverse ratio $\frac{maxRetDist}{minRetDist}$ as heuristics.

Null Pointer Checks: `null` (based on Section 3.2.3) Similar to early returns, we noticed that compilation units with many `null` checks may benefit from compilation. Thus, we determine the heuristics score by taking the number of `null` comparisons and increment it by one, `nNullCmp + 1`.

Table 3.2: Compilation unit features and their heuristic score functions.

Name	Letter	Heuristic Description	Score Function
ecyc	e	number of estimated CPU cycles per size	$cu \rightarrow \log\left(\frac{nCycles}{compiledSize}\right)$
init	i	compilation unit is a static initializer	$cu \rightarrow \begin{cases} 0: & cu \text{ is stat. init.} \\ 1: & \text{else} \end{cases}$
pars	p	number of parameters	$cu \rightarrow nPars + 1$
null	n	number of pointer comparisons against null [114]	$cu \rightarrow nNullCmp + 1$
rndX	-	random heuristic (fixed seed)	$cu \rightarrow rnd(1.0, 10.0)$
rtrn	r	inverse early return ratio	$cu \rightarrow \frac{maxRetDist}{minRetDist}$

Static Initializer: `init` (based on Section 3.2.4) For static initializers, we saw and argued that they may not benefit from compilation, since they are often rather large, executed at most once, but a majority is never executed.

Since static initializers are identifiable by their special name, our heuristic function simply returns 0 when it is a static initializer, and 1 for all other compilation units.

Random: `rndX` (control group) With the hybrid execution system as described in this paper, removing almost any compilation unit reduces binary size, but might decrease performance.

To identify how reliable the heuristics are and whether they generalize from *fop* to the complete benchmark set, we compare them against a random heuristic. Thus, they have to do better than selecting compilation units randomly for interpretation.

The heuristic assigns every compilation unit a random score from 1.0 to 10.0. To avoid comparing against only one randomly good or bad control, we use three different seeds and have the heuristics `rnd1`, `rnd2`, and `rnd3`.

3.4.3 Combined Static Heuristics

Combining single features into more complex heuristics often leads to better results [19, 114].

We combine a set of heuristics H into a new heuristic by multiplying the single score values as in Equation (3.2). Because of the multiplication, we can ensure that a single 0 score leads to automatic interpretation regardless of other score values. This way, we can make sure that static initializers are always interpreted if that heuristic is included.

$$heur_H(cu) = \prod_{h \in H} heur_h(cu) \quad (3.2)$$

Table 3.3: Sample compilation units and their scores for ipe heuristics.

cu	init: $\text{heur}_i(\text{cu})$	pars: $\text{heur}_p(\text{cu})$	ecyc: $\text{heur}_e(\text{cu})$	ipe: $\text{heur}_{ipe}(\text{cu})$
A	0	1	3	0
B	1	1	4.1	4.1
C	1	3	5	15
D	1	2	1	2

For simplicity, the name of combined heuristics consists of the letters of its single-feature heuristics. As an example, the *iprn* heuristic considers static initializers (*i*), the number of parameters (*p*), the early return ratio (*r*), and number of null checks (*n*). The score function of *iprn* is therefore defined as in Equation (3.3).

$$\begin{aligned} \text{heur}_{ipe}(\text{cu}) &= \text{heur}_i(\text{cu}) \cdot \text{heur}_p(\text{cu}) \cdot \text{heur}_e(\text{cu}) \\ &= \text{init}(\text{cu}) \cdot (\text{nPars} + 1) \cdot \log\left(\frac{\text{nCycles}}{\text{compiledSize}}\right) \end{aligned} \quad (3.3)$$

Table 3.3 shows four sample compilation units A to D. A is a static initializers, thus gets the value 0 assigned, and is therefore always interpreted. One can also see that the logarithm is important for the number of estimated cycles per size, *ecyc*: Without the logarithm, it would fully dominate the combined heuristic: Its values for heur_e would be 1000 and 10000 instead of 3 and 5, which would overrule all other single heuristics.

Combining features can help mitigate wrong decisions based on single outliers. Concretely, for the *iprn* heuristic, a 20% image size reduction leads to a performance slowdown of 8%, while single features cause slowdowns between 8% and 20% (more details in Section 3.6).

3.4.4 Dynamic Heuristic

Besides static knowledge of research question 1, we are also interested in research question 2: There are many compilation units that are not called at all or do not significantly impact the overall performance. For example, Section 3.2.4 proposed that static initializers, i.e., compilation units that are large but rarely or even never called, can be interpreted without significantly impairing overall program performance. Thus, it would be worth in general to know how often a compilation unit is called during program execution. This also allows us to get closer to the largest size reduction possible without impacting performance.

To include dynamic information as well, we suggest a heuristic combining the number of estimated cycles per byte (as in Section 3.2.3) with the call frequency, as shown in Equation (3.4):

$$\text{heur}_{\text{dyn}}(\text{cu}) = \frac{\text{nCalls} \cdot \text{nCycles}}{\text{compiledSize}} \quad (3.4)$$

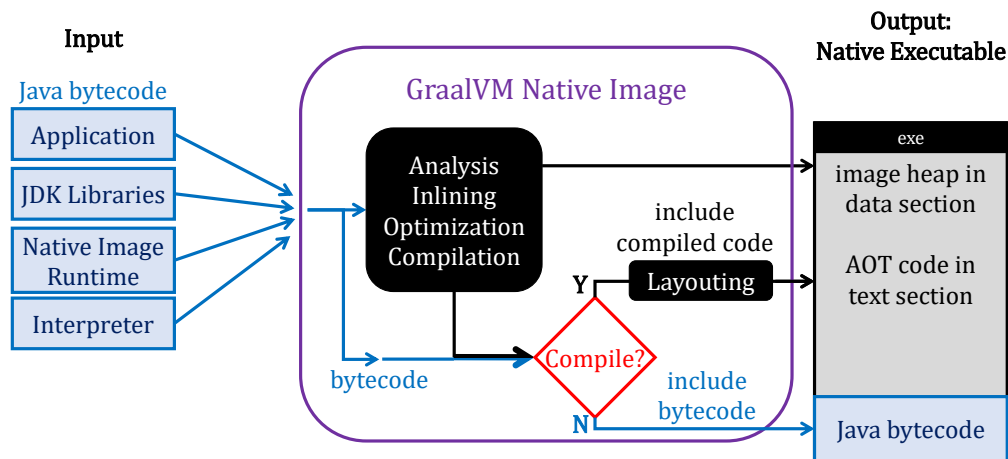


Figure 3.11: Native Image Build for Hybrid Execution.

3.5 Implementation

The approach described above is implemented for applications that can be compiled to Java bytecode. More concretely, it is applied to an AOT compiler for Java bytecode, GraalVM Native Image. While the basics of GraalVM Native Image have been described in Section 2.6.8, the following sections describe the implementation of this approach within GraalVM Native Image.

3.5.1 GraalVM Native Image

To enable a hybrid execution approach in GraalVM Native Image, we modified the compilation process as shown in Figure 3.11: Application code, JDK libraries and the Native Image Runtime are passed to the Native Image compiler. Similar to the general Native Image compilation toolchain procedure described in Section 2.6.8, the code is analyzed and optimized. One of these optimizations is inlining, where code of callees is embedded into their callers' code. If methods are compiled to machine code, a method and all its inlined callees are compiled together. This is called a *compilation unit*.

The approach described in this chapter requires a decision whether to compile or interpret code. Since GraalVM Native Image aggressively optimizes within a compilation unit and makes use of powerful inlining heuristics [110], we consider it reasonable not to separate code within a compilation unit. Therefore, a whole compilation unit is executed in the same way (either interpreted or AOT-compiled).

While the heap snapshot is written into the executable after analysis and optimization, there is one additional step introduced for code: The general Native Image toolchain would now take the compiled version of every compilation unit, lay it out, and store it into the executable. Since, however, the hybrid approach described in this chapter also offers interpretation, the toolchain can now replace the native code by the original Java bytecode, which is shown with a blue border in Figure 3.11. To be able to execute Java bytecode at run time, an interpreter is needed (which is not present in a Native Image executable by

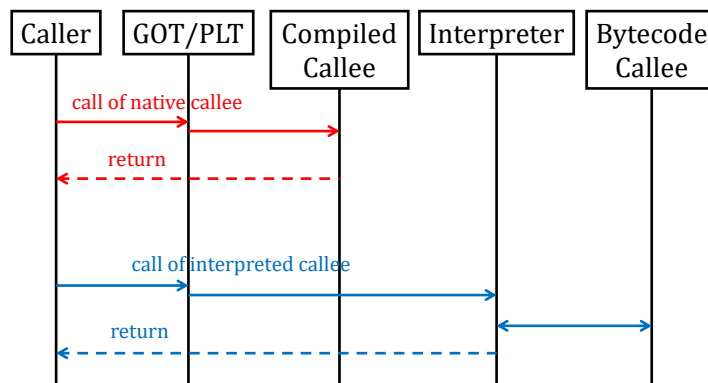


Figure 3.12: Indirection step via GOT/PLT allows dynamic resolution of method calls.

default). Therefore, a simple and light-weight interpreter, which can also be used for debugging (as described in Section 2.6.8), is added to the code to be compiled at the start of compilation.

3.5.2 Triggering Interpretation of Compilation Units from Native Call Sites

When a compilation unit supposed to be interpreted is invoked, it is not possible to simply continue execution from Java bytecode. Instead, a little stub has to start the interpreter and trigger interpretation of that compilation unit. In the following, we explain how to achieve such a call forwarding.

In a native executable generated by GraalVM Native Image, method calls between compilation units can use an indirect calling mechanism rather than direct jumps. A procedure linkage table (PLT) together with a global offset table (GOT) is available, which have been described in Section 2.4. Those tables serve as an intermediary lookup structure between callers and callees [41] and offer patching at run time. When the executable is initially loaded, the GOT is initialized with the addresses of the corresponding compiled methods. Call sites of a compilation unit target the corresponding PLT entry that accesses the pre-resolved GOT address. A simple call is shown at the top of Figure 3.12.

This GOT/PLT indirection can be used to trigger interpretation from native execution: Instead of the address to the AOT-compiled method, the GOT entry contains a trigger of the interpreter to interpret the target method. This procedure is shown at the bottom half of Figure 3.12.

Our current implementation uses the address to the PLT entry for every method call (in the interpreter as well as in native code). This means that an interpreted compilation unit can call the compiled version of a method. In case an interpreted caller calls a callee which is about to be interpreted as well, the interpreter calls the PLT entry stub, which lets execution continue in the interpreter, interpreting the bytecode of the callee. While this is an even slower way of realizing hybrid execution, interpreted code is not on the hot path anyway. Therefore, this performance penalty is accepted, while making a point to keep the

system small and simple, since the call-site does not need to know if the callee is compiled or meant to be interpreted.

The main reason for this indirection for every call, which comes with a slight performance drop, is research flexibility: Execution can be redirected to the interpreter or back to compiled code at run time, i.e., without having to recompile the native executable again.

3.5.3 Execution Mode Restrictions

As mentioned in Section 2.6.8, runtime code and application code are merged into a single compiled executable. This means, one cannot easily determine whether a given method in such an executable is an application method, a runtime method, or even both. However, as soon as compilation units are needed to implement the interpreter itself (independently whether they are used by the application code as well in a different context), they must always be executed in native mode. Otherwise, endless recursion will occur, as the interpreter would start to interpret itself. Additionally, there is VM code (e.g., from the garbage collector) that cannot be interpreted as it would end in a recursion too.

3.5.4 Performance-Critical Compilation Units

In applications, a single compilation unit can be critical for the overall performance, which we show with an example in the following:

We built a small sample application that organizes data in linked lists, but does not contain many other application methods.

The result of randomly replacing compiled code with its bytecode for that application is shown in Figure 3.13. Every dot in this Figure corresponds to one scenario of the application with certain code compiled, and certain code interpreted. The reason why there are so many compilation units is that during compilation, code of the JDK and the runtime is added, as described in Section 2.6.8. The x coordinate of every dot denotes the savings in code size, so a higher x value denotes a higher ratio of interpretation. The y coordinate represents the total execution time of the application in the given scenario.

Although such a linked list application is mainly for illustration, it clearly shows that there are only a few compilation units that impact performance. Whenever a critical compilation unit is removed, there is a significant performance drop. Such a removal of a critical unit is marked with a red rectangle in Figure 3.13. In that situation, the `LinkedList::add` method has been switched from compilation to interpretation. On the other hand, most of the still included compilation units do not change performance significantly, as shown with the dashed black rectangle. One reason for performance-irrelevant compilation units is that there are units that are not called at all.

3.5.5 Imperfection of Heuristics: Need for Run-Time Compilation

We use heuristics to automatically select performance-relevant compilation units for compilation, while other compilation units are interpreted. These

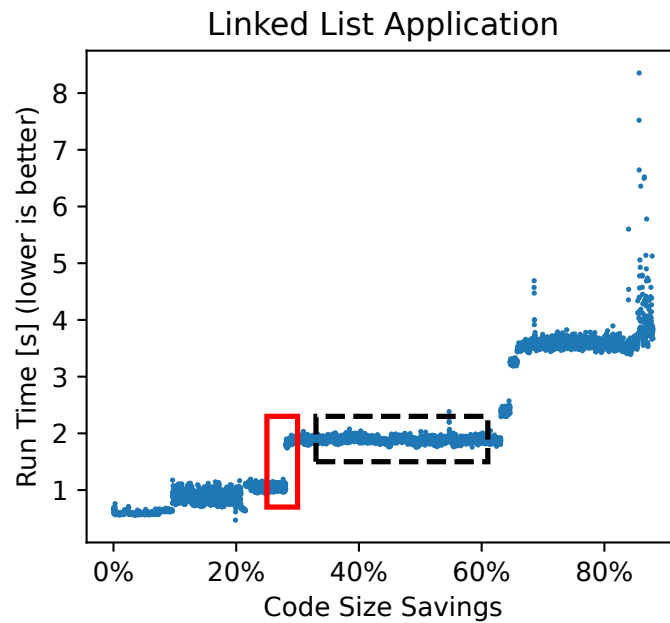


Figure 3.13: Scatter plot of compilation units of an application where only a few are performance-relevant. With a higher x value, more units are interpreted. Critical units can be detected when the run time increases, as marked in red.

heuristics are bound to mispredict the performance effects of interpreting single compilation units, as heuristics are not perfect. However, the example in Figure 1.1 clearly shows that such a misprediction can slow down the overall performance significantly.

To deal with such compilation units that cause a significant overall slowdown when interpreted, a compiled version of those compilation units should be executed instead. This is a classical task for a JIT compiler.

Using a JIT compiler on the frequently executed methods to improve peak performance would lead to a regular JIT compiler with a high warm-up time. In contrast, the aim of this approach is to JIT-compile as few compilation units as possible. Going back to the example shown in Figure 3.13, we assume that JIT-compiling just a single compilation unit (related to the jump marked by the red rectangle) would improve the overall result.

In the current implementation, there is no JIT compiler available, as GraalVM Native Image only offers a lightweight interpreter. Therefore, to receive compiled code for critical compilation units, one can rather replace the interpreted compilation unit by its AOT-compiled machine code instead. Although only few compilation units need to be replaced, a real JIT compiler comes with a drop in warm-up performance due to run-time compilation.

To find out compilation units that spend a lot of time in the interpreter, we performed the following measurements: We pick one compilation unit after each other, interpret it instead of executing its AOT-compiled code, and then track the interpreter time the compilation unit spent in the interpreter. Also, we track the total time. After 5 runs, the mean of these runs was taken. This is repeated

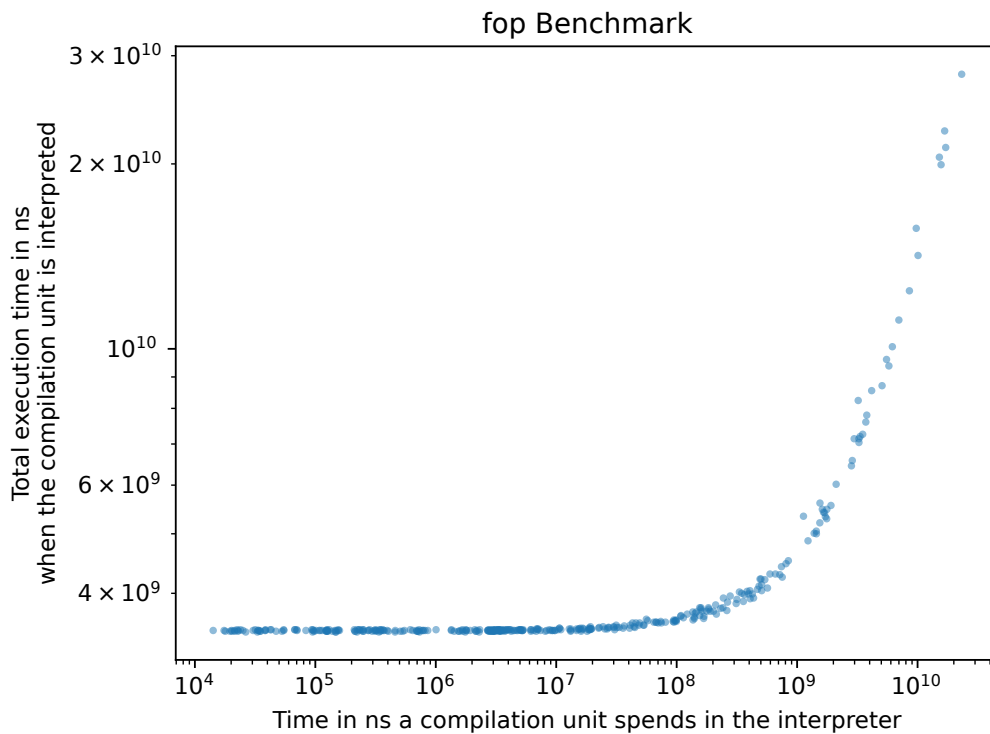


Figure 3.14: Comparison: Time spent in the interpreter vs. total execution time, each dot is a compilation unit. Starting from 10^8 ns in the interpreter, interpretation time also affects total execution time of the benchmark.

for every compilation unit. Then, we compare the time spent in the interpreter with the total time the benchmark ran.

The results are shown in Figure 3.14, which is a comparison of the interpreted time of a compilation unit against the total execution time when the compilation unit is interpreted. Now, again every dot represents one run of the application. In contrast to Figure 3.13, now only one compilation unit is interpreted at a time. While the y axis again denotes the total execution time of the application, the x axis now denotes the time of the interpreted compilation unit.

It can be clearly seen that up to $10^8 \text{ ns} = 100 \text{ ms}$ of interpreted time, there is only a minor impact on total execution time. Since measuring the self time of a compilation unit comes with profiling costs, one can take a look at the call density instead: If the time between two consecutive calls of an interpreted method is shorter than 100 ms , the bytecode of the method is replaced by its native code. In case of a real JIT compiler, the method would have been scheduled for JIT compilation.

3.5.6 Constant Overhead: Sizes of Interpreter and JIT Compiler

The main approach in this chapter as described above removes compilation units from the executable and replaces them with Java bytecode. While it reduces the size of the resulting executable, it also introduces a constant over-

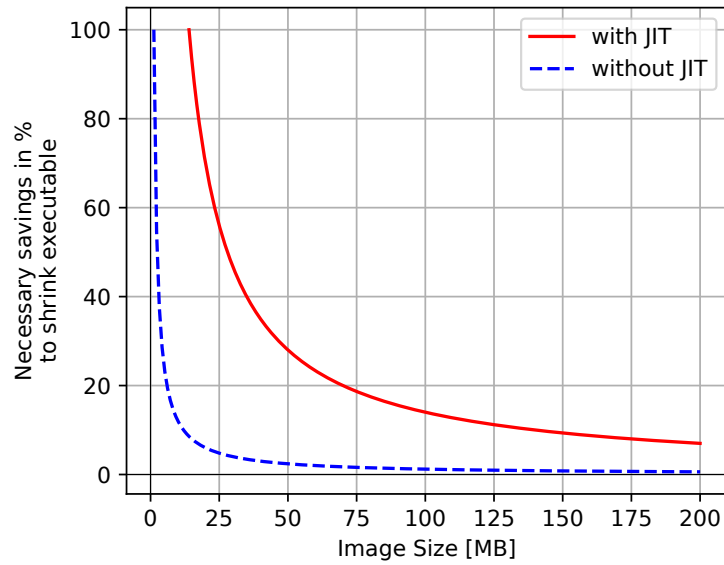


Figure 3.15: Necessary size savings by code removal to achieve smaller executables when interpreter and a JIT compiler are included.

head, as the interpreter and a JIT compiler have to be available during execution as well. While the size of the compiled interpreter code only consists of 1.1 MB, the code of a JIT compiler adds 12.8 MB to the executable. Thus, when integrating the interpreter into the executable, the current approach adds a constant overhead of almost 14 MB to the executable. As a consequence, applications smaller than 14 MB cannot be shrunk by this approach at all.

Concrete numbers can be seen in Figure 3.15: The x axis denotes the size of a Native Image executable without our approach. The y axis denotes the corresponding percentage values of code size savings our approach has to achieve to result in an executable file of the same size. While a 28 MB application needs to reach 50 % savings by code size to make our approach pay off, a 140 MB executable can be shrunk if more than 10 % of code size are saved.

Without using a JIT compiler, the savings are different, as shown by the dashed line in Figure 3.15. However, as described in Section 3.5.4, significant performance slowdowns may not be avoided.

A possible solution is to provide a system-wide shared library, rather than to compile the interpreter and the JIT compiler into each application. Therefore, if not stated otherwise, measurements within this chapter do not include the size of the interpreter and JIT compiler.

3.6 Evaluation

We evaluate the proposed heuristics by assessing their impact on the performance on 5 DaCapo and 13 Renaissance benchmarks. Since there is a direct tradeoff with the number of methods selected for interpretation, we assess the performance impact when aiming for 10%, 20%, and 30% reduction of the

overall image size. We first assess the heuristics individually and then their combination.

3.6.1 Methodology

We performed the measurements on a system with an Intel Core i7-1255U CPU, 32GB RAM and Ubuntu 24.04.4 LTS using Linux Kernel 6.17.0. During measurements, turbo boost and network connections were turned off. We used GraalVM 25 as of 2025-09-16. Each benchmark was run 10 times for each of the setting, with 50 iterations of the core benchmark. Thus, per benchmark and benchmark setting, we collected 500 data points to account for system noise. Since our system uses ahead-of-time compiled and interpreted code only, we did not specifically account for warmup, but did not see large variations in the data.

Assessed Heuristics For our evaluation of the proposed heuristics, we compare them to a baseline where all code is AOT-compiled. As previously mentioned, we also compare to three heuristics that rank the compilation units selected for interpretation randomly. To account for a possible impact on different seeds, we include three random heuristics `rnd1`, `rnd2`, `rnd3` as control group.

We assess the single-feature heuristics `ecyc`, `pars`, `rtrn`, and `null` (see Table 3.2). Since `init` is a binary classifier, we did not evaluate it as its own heuristic against a desired size reduction because by itself it does not give a ranking.

To assess the combination of heuristics, we combine the 5 subsets of 4 heuristics each as well as the combination of all 5 heuristics as described in Section 3.4.3. Thus, we assess `iern`, `ipen`, `iper`, `iprn`, `pern`, and `ipern`. Their names contain the letters of the single-feature heuristics as shown in Table 3.2.

Afterwards, we also evaluate the dynamic heuristics described in Section 3.6.4.

Interpreter Size and Execution Mode As described, our hybrid execution replaces AOT-compiled code by Java bytecode, as shown in Figure 3.11. Thus, execution also requires a lightweight, AOT-compiled interpreter. On x86-64, it is about 1.1 MB of size, an optional JIT compiler would be about 12.8 MB. We exclude this from our size measurements because it can be provided as a system-wide shared library rather than compiled into each application.

As single wrong decisions by our heuristics can strongly impact performance negatively in case a performance-relevant compilation unit is interpreted, the compiled code replaces the Java bytecode at run time if a compilation unit is called twice within 100ms. In our experiments, only $\approx 2\%$ of all interpreted units are replaced.

Benchmarks The Java interpreter that we use in Native Image is a research prototype. Together with the closed-world assumption of Native Image, it unfortunately limits the currently supported benchmarks. This means, we can only use `fop`, `h2`, `pmd`, `sunflow`, and `xalan` benchmarks from DaCapo,

and akka-uct, finagle-chirper, finagle-http, fj-kmeans, future-genetic, par-mnemonics, philosophers, reactors, rx-scrabble, scala-doku, scala-kmeans, scala-stm-bench7, as well as scrabble from the Renaissance benchmark suite.

3.6.2 Results for Single-Feature Heuristics (research question 1)

Figure 3.16 shows the results for individual heuristics as well as random heuristics. The x axis shows the slowdown relative to the baseline. The results are grouped by overall size reduction we aimed for, 10%, 20% and 30%, respectively.

These results indicate that the number of parameters (pars) is the best single-feature heuristic. It allows us to reach the desired size reduction with the lowest median increase in run time over all benchmarks. It achieves a size reduction by 10% with a slowdown of only $\approx 4\%$ at the median, a 20% reduction with 8% slowdown, and a 30% size reduction with a 11% median slowdown.

The next best heuristics are the early return ratio (rtrn) and the estimated CPU cycles per byte (ecyc). The null pointer check heuristic performed worst among our single-feature heuristics. Only the three random heuristics rnd1 to rnd3 performed worse, which is encouraging, since this shows that all heuristics perform better than a random selection of compilation units.

3.6.3 Results for Combined Static Heuristics (research question 1)

The results for combined heuristics are shown in Figure 3.17. Again, the results are grouped by size reductions – 10%, 20% and 30%, respectively. Over all benchmarks, one can see that our combined iper heuristic, relying on static initializers, number of parameters, number of estimated CPU cycles per size, and early returns, yields the best results. For size reductions of 10%, 20% and 30%, the slowdown is only 2%, 4% and 6%, respectively. This means it also improves over only using the number of parameters, the best single-feature heuristic.

Furthermore, the results indicates that the null pointer check heuristic is not beneficial for these benchmarks, the combinations that include it perform worse than iper.

Overall Trade-off Between Performance and Size

As the main trade-off of interest to us is the tension between reducing code size without impacting performance significantly, we investigate it in more detail for the most promising iper combination of heuristics.

Figure 3.18 shows it for size reductions from 10% to 35% in 5% increments. As expected, a smaller executable leads to an increase in execution time. Interestingly, we seem to be able to reduce the native image size by over a third at a cost of less than 20% performance for the slowest benchmark we measured and the median slowdown stays under 7%.

Figure 3.19 plots the medians for the size reduction targets between 10% and 35%. Here we also include the random heuristics, though, they drop off the

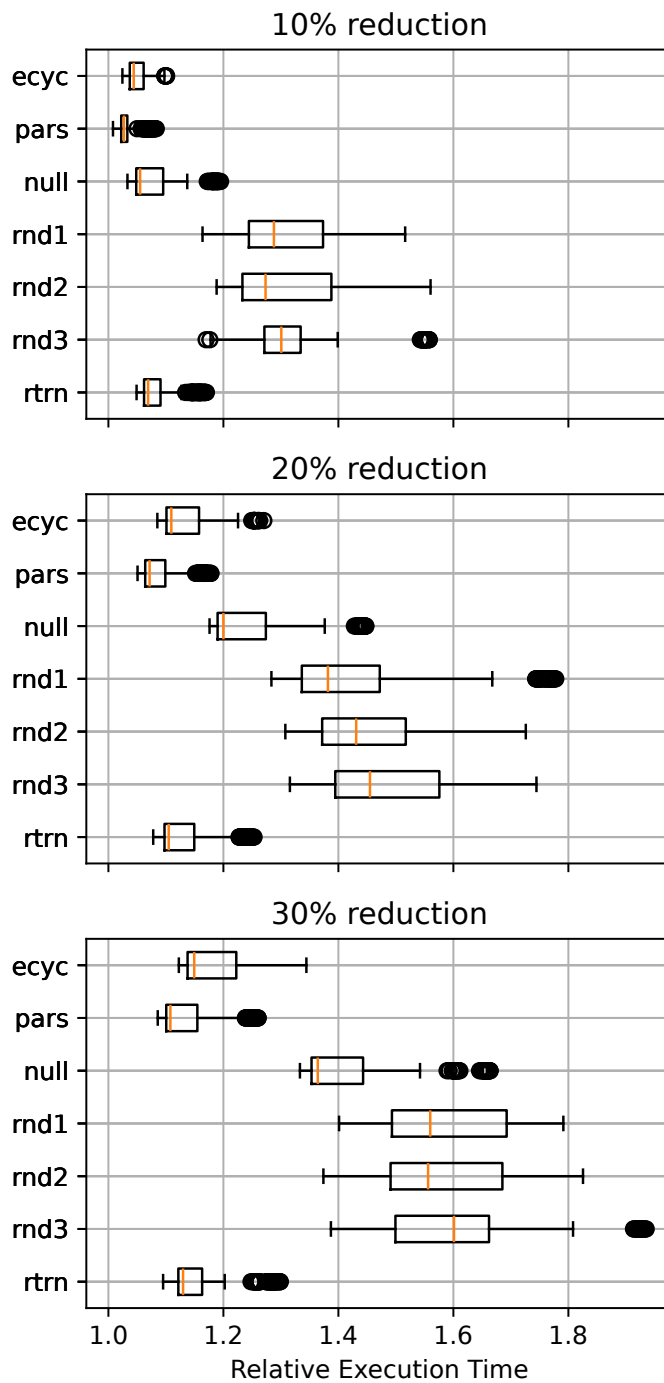


Figure 3.16: Single-feature results, grouped by size reduction and heuristics. The heuristic based on number of parameters (pars) achieves the size reduction with the lowest slowdown, and thus, is the best individual heuristic.

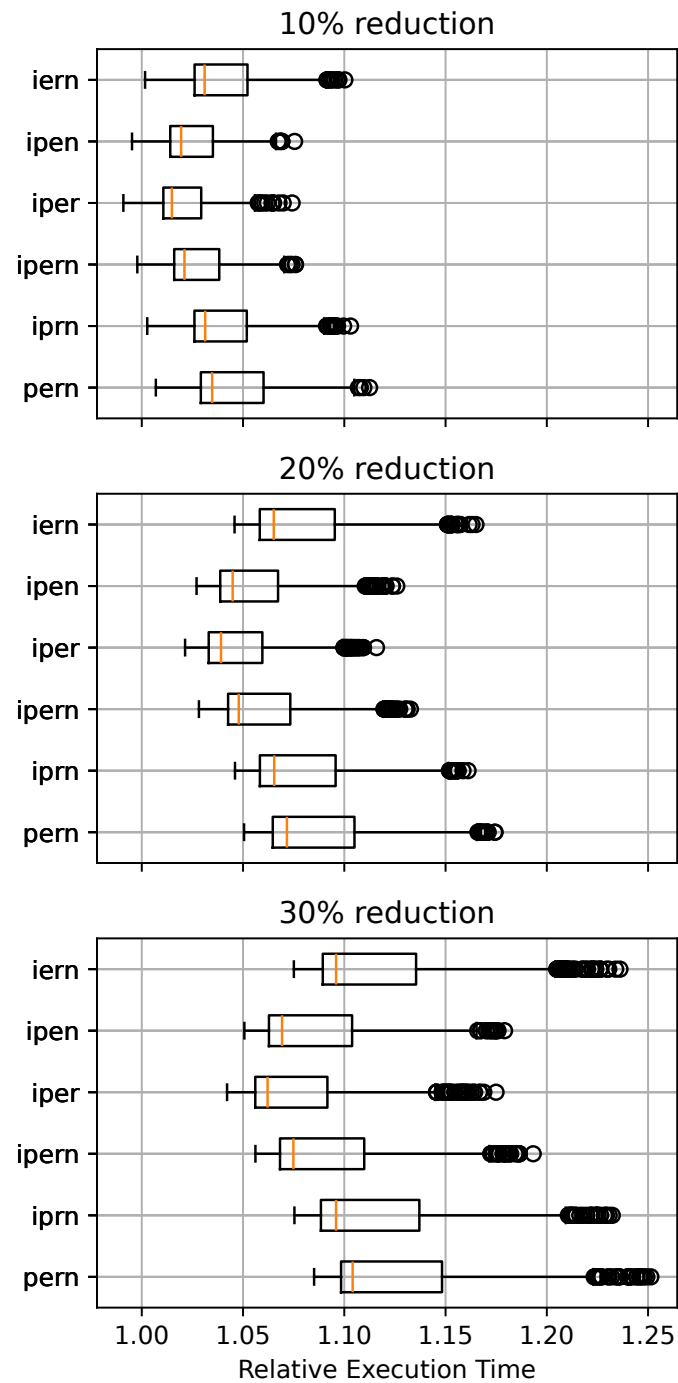


Figure 3.17: Combined results, grouped by size reduction and heuristic. Overall, the iper combination of heuristics achieves the size reduction with the lowest overhead.

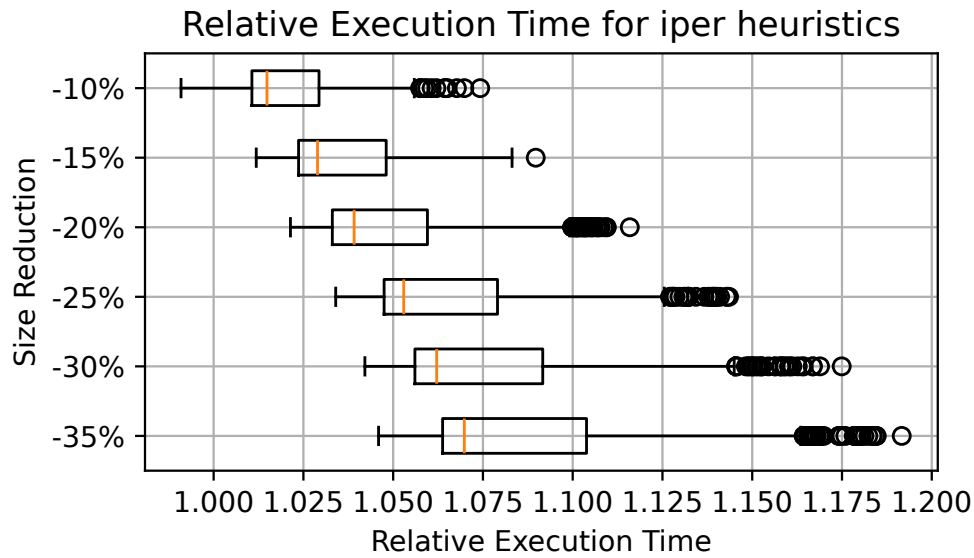


Figure 3.18: Trade-off between size and performance for our iper heuristics. At a size reduction of 35% we see a worst case slowdown of less than 20% with the median slowdown staying under 7%.

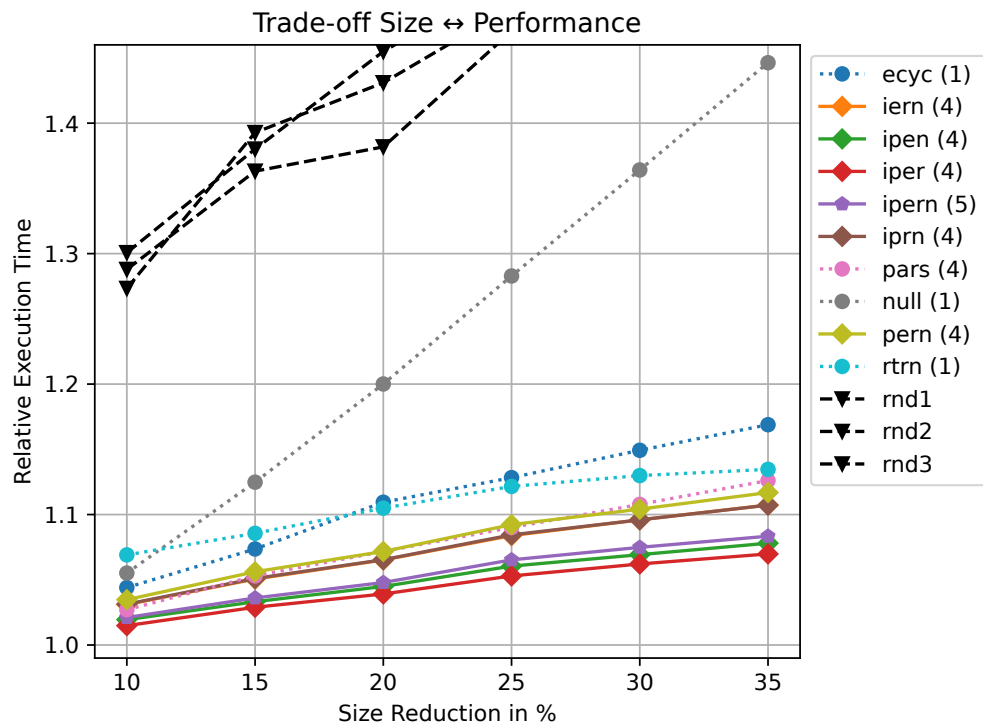


Figure 3.19: Trade-off between size and performance per heuristic. Showing the median slowdown. iper is the overall best candidate, while the random heuristics are causing the highest slowdowns overall.

Table 3.4: Native Image Executables of DaCapo Benchmarks.

Benchmark	Size	#Compil. Units	Exec. Time
fop	180 MB	59,241	77 ms
h2	68 MB	45,086	2945 ms
luindex	50 MB	18,411	390 ms
lusearch	45 MB	18,539	349 ms
pmd	95 MB	63,850	1385 ms
sunflow	37 MB	19,316	1045 ms
xalan	62 MB	48,395	244 ms

chart fairly quickly as they cause high slowdowns. The null pointer check heuristic is overall causing the worst slowdown of all our heuristics. While *iper* is the best candidate, *ipen* and *ipern* are only slightly worse, likely because of the negative contribution of the null pointer check heuristic.

Impact on the Individual Benchmarks

As the box plots already indicated, the impact on individual benchmarks can differ significantly. Thus, Figure 3.20 shows the results for the benchmarks using our most promising heuristics *iper* with a target size reduction of 20%.

On average, the Renaissance benchmarks perform better than the benchmarks of the DaCapo suite. The ray tracing DaCapo benchmark *sunflow* seems to have a similar structure to the Renaissance benchmarks. In our experience, these benchmarks are smaller than some of the other DaCapo benchmarks.

The DaCapo *fop* benchmark, on the other hand, shows the highest performance overhead with our proposed heuristics. This is interesting, since the *fop* benchmark was the one we analyzed in Section 3.2 to identify candidate heuristics. We consider this a good indication that their heuristics may generalize well to other Java-based applications.

Similar to the *fop* benchmark of DaCapo, also both *finagle* benchmarks of Renaissance show a high performance overhead. A possible reason for that these benchmarks contain often-called but short-living closures, which leads to a high allocation and collection rate [108]. While the analysis of GraalVM Native Image seems to be able to solve this issue by analysis, interpretation leads to the slowdown described above.

3.6.4 Results for Dynamic Heuristics With and Without Simulated JIT Compilation (Research Question 2)

Sizes, compilation units and wall-clock times of the Native Image executables are shown in Table 3.4.

Each of the following scenarios was performed in the same way: First, we compiled the benchmarks with GraalVM Native Image and did a run where everything is executed natively (which produces the numbers in Table 3.4). Then, we

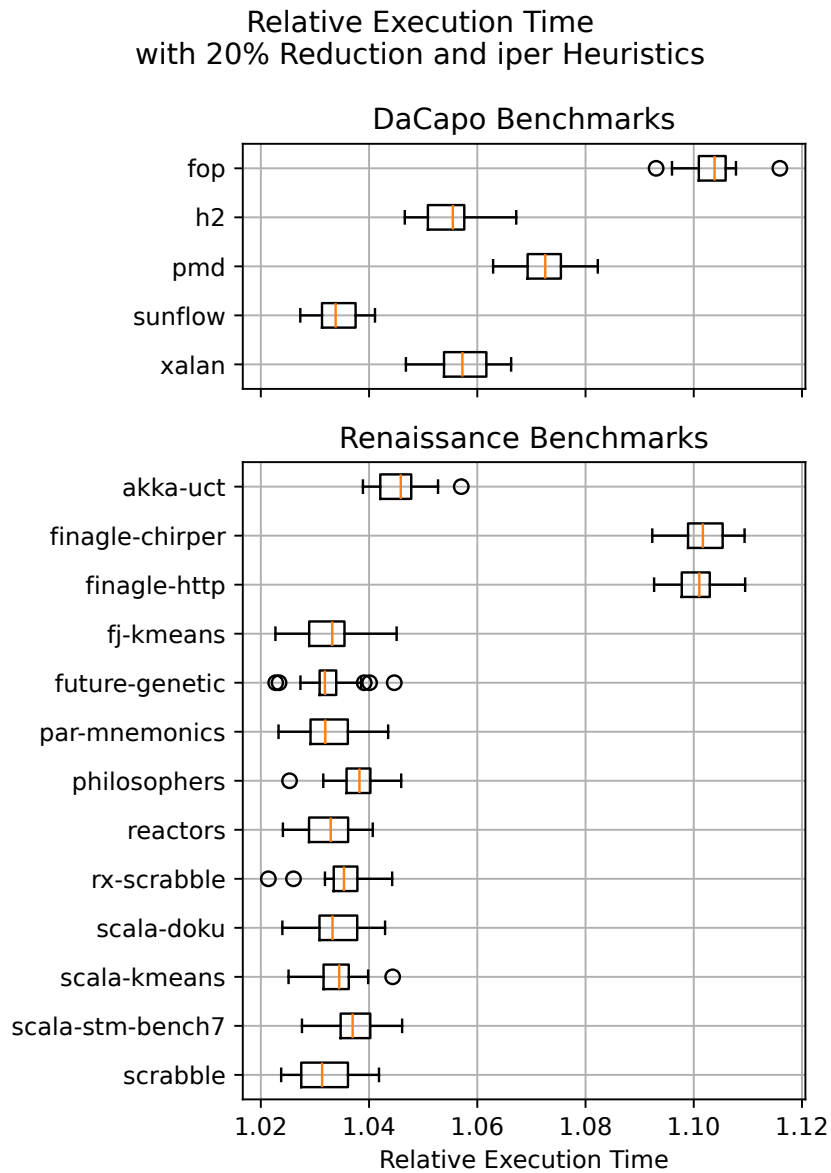


Figure 3.20: Detailed benchmark results for a 20% size reduction using the iper heuristic. The highest slowdown is caused on fop, while most Renaissance benchmarks see only minor slowdowns.

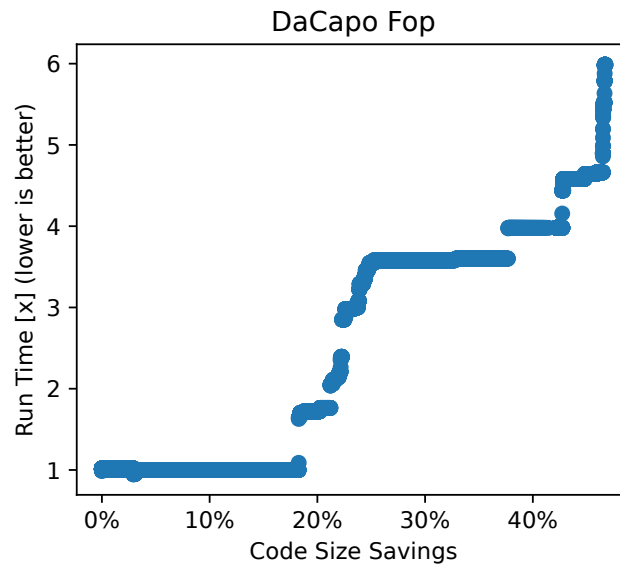


Figure 3.21: Run with sorted order and dynamic heuristic – DaCapo fop.

removed subsets of compilation units from the executables, i.e., we let them be interpreted instead, and tracked the performance in separate benchmark runs. Additionally, when an interpreted compilation unit is called twice within 100ms, we simulate a JIT compilation as mentioned in Section 3.5.4.

The results are shown in scatter plot diagrams (Figure 3.22): The x-axis denotes the ratio of the size of removed compilation units (from 0 to 100 %). Thus, 0 means that everything is AOT-compiled. Due to code that can only be AOT-compiled as described in Section 3.5.3, not everything can be interpreted. This is why the x axis stops before 100 %.

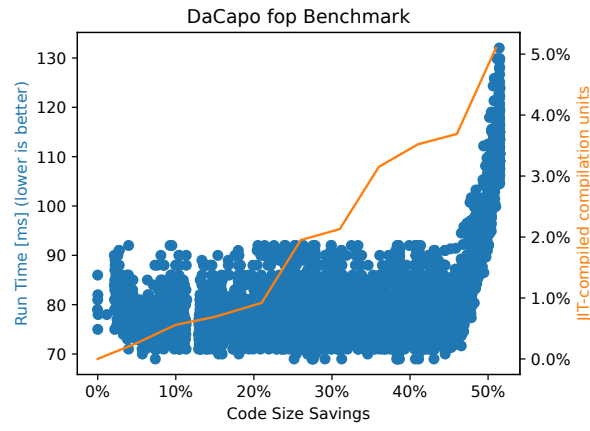
For the y-axes in Figure 3.22, two different metrics are plotted: The left y-axis shows the peak execution time. The right y-axis shows the ratio of compilation units that triggered our simulated JIT compilation if enabled.

The measurements in the individual benchmarks were stopped as soon as either peak performance reached the limit of 2 x, or the ratio of compilation units triggering simulated JIT compilation exceeded 5 %.

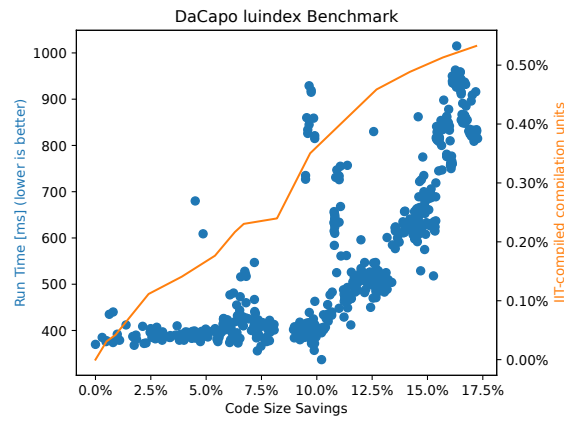
We show two scenarios in the following. For both runs, the compilation units are sorted by our dynamic heuristic defined in Section 3.4.4

- **Sorted selection without JIT:** For this run, we simply follow the heuristic — compilation units are either compiled AOT, or interpreted upon called.
- **Sorted selection with JIT:** For this run, we add the option of our simulated JIT compilation as described in Section 3.5.4, where the AOT-compiled code can replace the interpreted bytecode at run time.

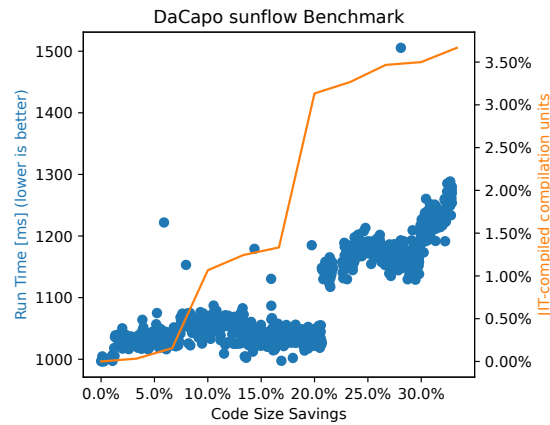
Sorted Selection Without JIT For this run, we sorted the compilation units by call frequency and the number of estimated cycles per byte. This sorting order requires a single profiling run to determine call data. In this scenario, the



(a) fop



(b) luindex



(c) sunflow

Figure 3.22: DaCapo Benchmarks with dynamic heuristic: Execution time and ratio of JIT-compiled code.

Table 3.5: Possible sweet spot results of DaCapo benchmarks, sorted by total image size changes.

Benchmark	JIT Ratio	Abs. Size Changes	Rel. Size Changes	Size Changes with JIT/Interpreter
DaCapo fop	4.1 %	−80.3 MB	−44.6 %	−36.9 %
DaCapo pmd	3.8 %	−27.6 MB	−29.0 %	−14.4 %
DaCapo h2	4.4 %	−21.2 MB	−31.2 %	−10.8 %
DaCapo xalan	3.8 %	−17.2 MB	−27.8 %	−5.4 %
DaCapo lusearch	1.9 %	−8.3 MB	−18.5 %	+12.4 %
DaCapo sunflow	3.7 %	−7.6 MB	−20.6 %	+17.0 %
DaCapo luindex	0.4 %	−5.2 MB	−10.3 %	+17.5 %

option of JIT compilation is disabled, i.e., we can only choose between AOT-compiled code and interpretation. The results for DaCapo fop are shown in Figure 3.21. On the one hand, removal of unused or rarely-executed code does not impact performance, and can lead to a code size reduction of $\approx 18\%$. On the other hand, removing one critical compilation unit, as happened at size ratio of $\approx 18\%$, highly affects the overall performance of the application. For the sunflow benchmark, the first critical compilation unit has been removed at 17% code size savings already, for luindex the first critical increase in peak performance is at $\approx 10\%$ of code savings. As it is not possible to have a heuristic that absolutely prevents the removal of critical code in the early stages, this scenario is bound to result in poor peak performance.

Sorted Selection With JIT For the third run, we add the option of our simulated JIT compilation as described in Section 3.5.4. Some of the detailed benchmark results are shown in Figure 3.22. The sweet spots of the smallest possible executable file while keeping peak performance are shown in Table 3.5. The *Size Changes* column denotes the changes of code savings. The right-most column, *Size Changes With JIT/Interpreter*, denote the size savings if the interpreter and the JIT compiler are included into the final executable, as discussed in Section 3.5.6.

While using the same sorting order as in the previous scenario, peak performance does not significantly change with respect to fully AOT-compiled code. The ratio of JIT-compiled code is indicated by the orange line in Figure 3.22, denoting the values of the right y-axis. We can see that this scenario offers a size reduction between 10.3% (luindex, Figure 3.22b) and 44.6% (fop, Figure 3.22a) without a significant effect on peak performance.

Also, the effect of our simulated JIT compilation is visible in Figure 3.22: The sunflow benchmark triggered 3.6% of the compilation units to leave the interpreter, which led to a 20% size reduction at the sweet spot. Only 0.36% of the compilation units triggered JIT compilation in the luindex benchmark. As (simulated) JIT compilation contributed less to improve peak performance in this scenario, the size of the executable file could only be reduced by 10.6% without a significant impact on peak performance.

Concerning warm-up performance, we cannot provide accurate results, as our

JIT compilation is only simulated. However, as we limited the ratio of JIT-compiled methods to 5 %, we expect a low impact on warm-up performance.

Reduction of Profiling Runs by Binary Search The diagrams shown in Figures 3.21 and 3.22 require many singular runs of the same application, where every single run is done with a different set of interpreted compilation units. While the sorting of the compilation units can be easily done at build time, a single run per compilation unit is clearly not feasible due to time and power consumption.

However, one can use the sorted compilation unit list and apply a binary search to find a suitable scenario. Instead of removing single compilation units from the executable, one can also remove (and add) buckets of compilation units at once. Thereby, only a few runs are necessary to determine a scenario close to the sweet spot. In case no profiling runs are possible, the static heuristics described in the previous section (Section 3.6.3) can also be used.

Total Size Savings Depend On Executable Size When applying our approach to an application, a set of compilation units is interpreted instead of AOT-compiled. This means that machine code is removed and replaced by its (in most cases) smaller Java bytecode, which leads to a size reduction of code. For the DaCapo benchmarks, these code size reduction numbers can be found in the third (absolute) and fourth (relative) column in Table 3.5. However, as mentioned in Section 3.5.6, there is also an overhead of ≈ 14 MB introduced by our approach, which consists of the interpreter and the JIT compiler that have to be added to the executable.

Figure 3.23 shows this overhead (as in Figure 3.15) together with the achieved code size reduction of the DaCapo benchmarks from Table 3.5. Additionally, the final size changes of the executables (including calculations for heap and code section for application code, interpreter and the JIT compiler) are shown in the last column in Table 3.5. Both, in Table 3.5 as well as in Figure 3.23, one can clearly see that our approach leads to better results for larger executables. On the other hand, an already small application might even become a larger executable file with our approach. This is the case for the sunflow benchmark, where a code size reduction of 7.6 MB, which corresponds to 20 %, leaves an overhead of 6.4 MB or 17 %. For larger executables, the overhead by the interpreter and JIT compiler is small enough to result in a significant overall size reduction. Concretely, the code size reduction for fop is 80.3 MB or 44.6 %, which changes to a reduction of 66.3 MB or 36 % after adding the interpreter and JIT compiler.

3.6.5 Comparing Static and Dynamic Results (Research Question 3)

To compare the effectiveness of static and dynamic heuristics and answer research question 3, we compare the most successful static heuristic *iper* with our dynamic approach of Section 3.6.4. While *iper* is based on static initializers, number of parameters, number of estimated CPU cycles per size and return ratio, the dynamic heuristics combines number of estimated CPU cycles per size

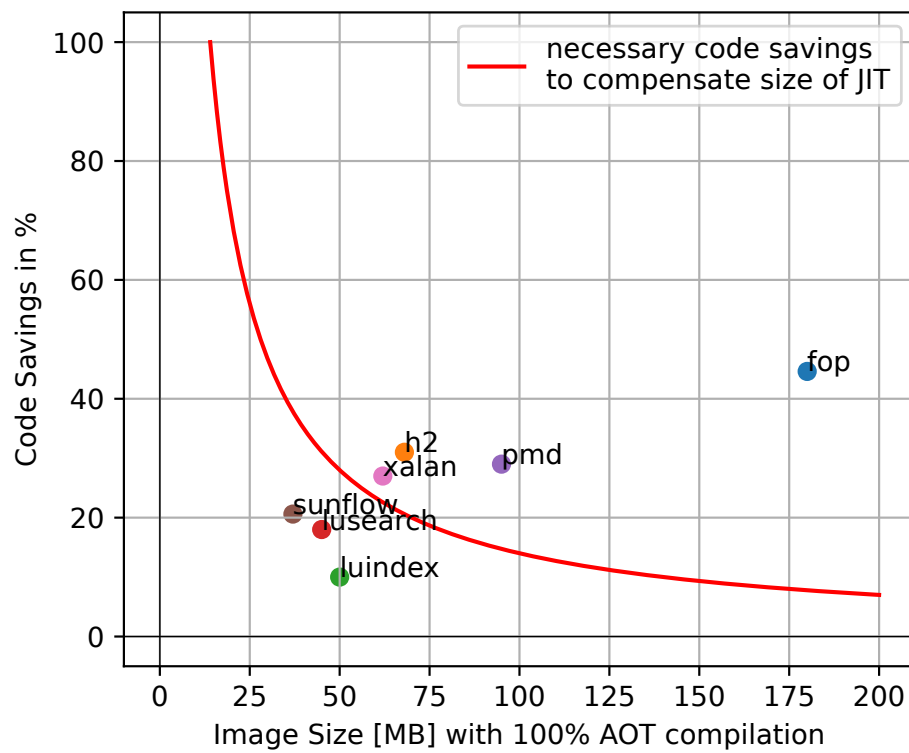


Figure 3.23: Achieved code savings of different benchmarks. A point above the line denotes a total saving of size for the corresponding benchmark.

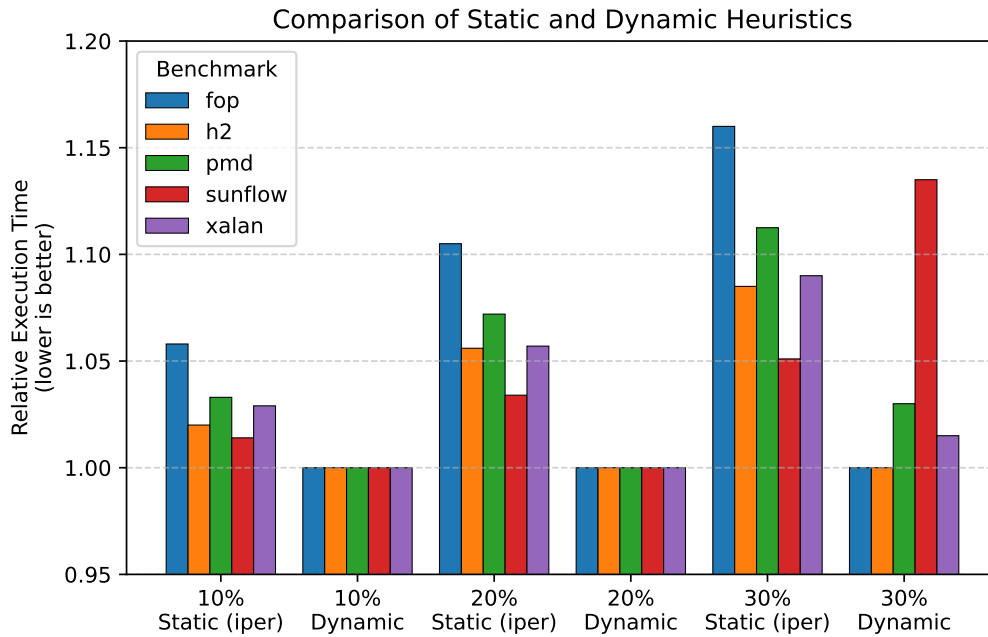


Figure 3.24: Comparing static and dynamic heuristics with a designated size reduction. Dynamic heuristics detect uncalled compilation units and thus perform better in most cases. They do not even cause a slowdown as long as uncalled compilation units are removed.

and the call count. For both, we replace compiled compilation units by their bytecode and take a look at peak performance, given the application is 10%, 20% and 30% smaller. As a baseline, we take an executable where all compilation units are AOT-compiled and executed natively.

The results are shown in Section 3.6.5. In most cases, our dynamic heuristic defined in Section 3.4.4 outperform the static *iper* heuristic defined in Section 3.4.3, which itself was the best among the static heuristics in our measurements. Since our dynamic heuristics are based on the activation count of compilation units, they treat uncalled units as interpretation candidates. Thus, size reductions of 10% and 20% incur no slowdown relative to the fully compiled application.

An exception occurs with the 30% reduction for the *sunflow* benchmark, where the static heuristics perform better than the dynamic ones. A plausible reason is that the dynamic heuristics rely on the estimated cycles and size (i.e., the static *ecyc* heuristic from Section 3.4.2) plus the call count. When performance-relevant units have similar call counts, static heuristics that combine multiple features can be more effective than our dynamic heuristics that depend on fewer factors. Nevertheless, our dynamic heuristics remain useful for reducing executable size without significant performance slowdown, as they primarily affect uncalled code. Consequently, expanding our dynamic heuristics to more factors (such as we did with the static heuristics) is sensible future work.

3.7 Limitations

Similar to many optimizations, the approach presented in this chapter may not always lead to a better result. The following circumstances may limit the effectiveness of the described approach.

Heuristics Provide Estimates The most critical aspect is the fact that heuristics only provide estimates. However, Figure 3.13 showed that interpreting a single, performance critical compilation unit can slowdown the whole application significantly.

Some Compilation Units Must be Compiled Section 3.5.3 mentioned that certain code has to be executed in native mode. However, due to the characteristics of GraalVM Native Image, it is not possible to determine for sure if code can be interpreted. This is because during analysis, GraalVM Native Image does not distinguish between application and runtime code. Therefore, it can happen that a runtime method that cannot be interpreted gets inlined into an application method. After this inlining step, the whole compilation unit cannot be interpreted, although it originated from the application code.

Warm-Up Performance We consider the impact on warm-up performance to be low, as there are only a few interpreted compilation units that are swapped back to native mode at run time (as described in Section 3.5.4). However, since we do not use a real JIT compiler in that situation, we are not able to provide measurements concerning warm-up performance.

Limited Build Time Savings As mentioned in Section 3.2.1, code is first compiled to machine code before we select whether to use the compiled code or interpretable bytecode of a compilation unit. On the one hand, we gain information for our heuristics from the performed analysis. On the other hand, analyzing and compiling code to a compilation result keeps build time high, even if our heuristics then decide to include the bytecode instead of the compiled code into the final executable. Thus, while our current approach saves binary size, it does not save build time. Regarding potential savings of build time, possible future work will be discussed in Section 7.1.

3.8 Summary

In this chapter, we presented the approach of reducing binary size of Native Image applications by replacing compiled code with its intermediate representation and adding a lightweight interpreter. While compiling and optimizing performance-critical code ahead-of-time (AOT), non-critical code is interpreted, which leads to a size reduction of the resulting binary executable. When performance-critical code is interpreted, which is bound to happen if heuristics are used, we showed that an option to switch to compiled code is necessary to keep peak performance high.

Research Question 1: What is the Trade-off Between Size and Performance When Applying Static Heuristics? Without profiling runs, static heuristic are able to achieve acceptable results, such as an average performance slowdown of 4 % for 20 % size reduction. Especially, static initializers turned out to be suitable candidates for interpretation. Independently whether dynamic or static heuristics are used, the application can be tuned along the trade-off between performance and size: Depending on the desired size reduction, one can further tune how much slowdown is acceptable.

Research Question 2: What is the Largest Possible Size Reduction of Applications Without Significantly Impacting Run-Time Performance? Dynamic heuristics that include the call count of compilation units yield the best results in the performed evaluation, but require run-time information and thus profiling runs in advance. Nevertheless, code that is never executed is an easy candidate for interpretation, i.e. to reduce size without affecting performance at all. Thereby, a size reduction of 44 % could be achieved without significantly impacting performance. In case the additional interpreter is also added to the executable (and not provided via a system-wide shared library), the size reduction in this case is still more than 36 % or 66 MB out of 180 MB.

Research Question 3: How Close do Static Heuristics Approach a Close-to-Optimal Solution Based on Dynamic Information? Dynamic information includes call counts for compilation units, enabling us to interpret code that is never executed. This reduces application size without affecting performance. Because call counts are unavailable without profiling, our static heuristics cannot match the peak performance of the dynamic approach. Consequently, as long as un-called code can be interpreted, the dynamic heuristic keeps peak performance of the baseline and clearly outperforms static heuristics. Nevertheless, static heuristics can still shrink binaries with only minor slowdowns, while requiring only low compilation-time effort at the same time. For the sunflow benchmark with 30% reduction, more complex static heuristics outperform our simple dynamic heuristic.

Chapter 4

Implementing Hybrid Execution of Native Extensions in a Multi-Language VM

The overall goal of this chapter is to continue exploring ways to mix execution modes to balance different compilation trade-offs. In this chapter, we present a hybrid execution approach for native extensions in multi-language, i.e., polyglot, applications. While native extension code can be compiled AOT and executed natively to reduce warm-up times, interpretation and JIT compilation via GraalVM offer mixing languages seamlessly.

Our goal is to combine the advantages of both, fast warm-up by AOT-compiled code, and high peak performance by cross-language optimization of GraalVM. Thus, it is necessary to implement a hybrid execution model for native extensions, such that they can be executed natively, as well as interpreted and JIT-compiled via GraalVM. To enable such a hybrid execution, for example, interpretation and compiled code have to be able to mutually redirect execution to the other mode. Additionally, since both execution modes use different heaps, it is also necessary to manage memory accesses between different execution modes. This chapter describes our solutions for such problems and how we implemented such a hybrid execution within GraalVM.

4.1 Motivation

Conventional multi-language (i.e., polyglot) applications are often executed in a managed runtime with, for example, garbage collection and JIT compilation, while computationally intensive code is written as a native extension. Usually, native extension code is written in a lower-level language that can be compiled to native code ahead of time. Popular examples are e.g. the Python packages NumPy or PyTorch [67], where their intensive computation tasks are written in C and executed natively. Mixing programming languages like this, as sketched in Figure 4.1a, however, requires developers to explicitly manage mode selection by strategically choosing languages for different code segments. The native extension in Figure 4.1a is written in C and compiled AOT. Functions of this native extension are denoted as black circles, and we call the execution of such functions *native*. Python code, or other code of a managed language such as JavaScript, is denoted with rectangular shapes. In this thesis, such code is

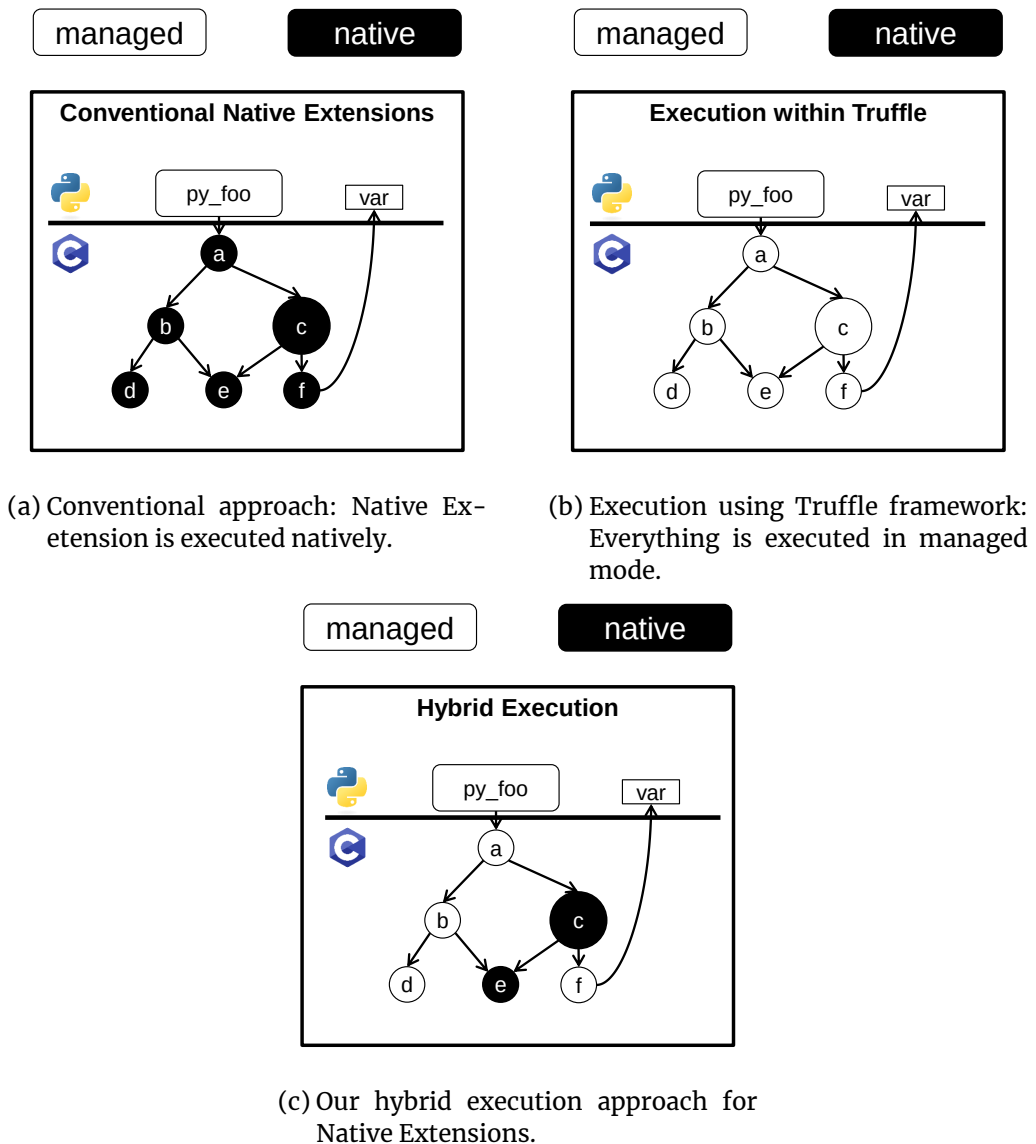


Figure 4.1: Applications with Native Extensions can be executed in different ways. Nodes denote functions, while arrows denote calls or accesses.

called *foreign* code, as from a native extension perspective, it is written in a foreign language. Foreign code is usually interpreted and/or JIT compiled. As already claimed, the programmer has to decide the execution mode by selecting in which source language to write the code. The boundary between these execution modes is also visible in Figure 4.1a by the line separating the languages.

To remove such a boundary between execution modes, the Truffle framework enables seamless optimization across programming languages, as described in Section 2.6.2 and as shown in Figure 4.1b. However, a system where everything is JIT-compiled comes with high warm-up times, especially when large amounts of data are processed by code which would have been executed natively in conventional polyglot systems. In such a case, also the code of the native extension is interpreted and JIT compiled. In Figure 4.1b, such code is denoted with white circles. We call the execution of such functions *managed* in

this thesis.

The idea to tackle this problem is to mix different execution modes for code of native extensions, such as sketched in Figure 4.1c. While certain parts of that code benefit from AOT compilation and native execution, such as conventional native extension code written in C, other code that is close to the language barrier might benefit from still being JIT-compiled, enabling optimizations such as inlining across languages. Thus, we want to mix both, managed and native execution, for native extensions.

An example for such a situation would be an application using the Python `lxml` package [56], which is an extensible markup language (XML) parser written in C. Executing all C code natively introduces a language boundary specified by the developers and users of the `lxml` library. This disables cross-language optimization and, for example, forces foreign objects accessed by the native extension to be converted. Executing all C code in managed mode, i.e., via Sulong, enables accessing foreign objects in their language, but introduces high warm-up times. Therefore, we aim for mixing managed and native execution for `lxml` code.

4.2 Basic Mixing Approach

The starting point for mixing managed and native execution for C code is to make use of the Sulong toolchain. As described in Section 2.6.6 and again shown in Figure 4.2, the Sulong toolchain produces native code with LLVM bitcode attached. If the source file contains a function call from a caller to a callee, both functions are present in both compilation results (LLVM bitcode and native code). Our basic approach is to redirect execution in a way such that e.g., always the native version of the caller is used, while the callee is always executed in managed mode (via Sulong).

The following subsections describe the different categories for necessary changes to enable such mixing of managed and native execution.

4.3 Heap Access

Since running C code on GraalVM can happen in different execution modes, there are also different possibilities for storing data. Native execution allocates and stores data in native memory, whereas GraalVM uses the Java heap to store objects. When functions that access the same object are executed in different modes, problems might arise depending on how the object is stored and accessed: Data stored in native memory might have to be accessed from managed execution, and vice versa.

4.3.1 Language Semantics Violation: Native Code Accessing Managed Objects

This subsection describes why accessing the managed heap from native execution may violate language semantics. Higher-level languages usually perform

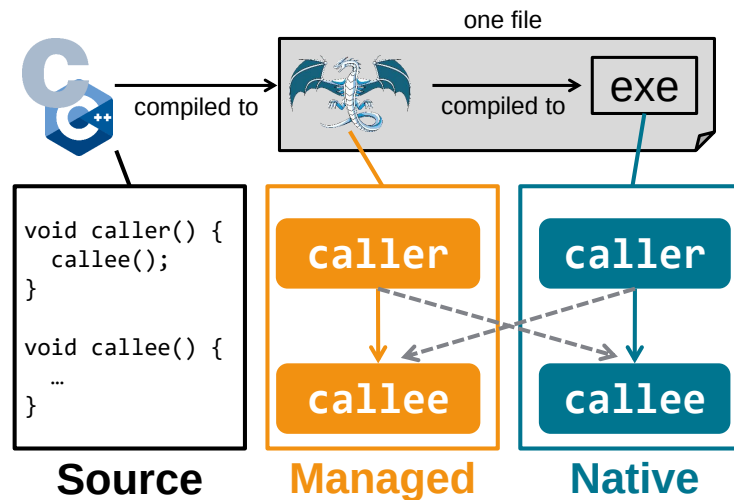


Figure 4.2: The compilation result of the Sulong toolchain includes LLVM bit-code for managed execution as well as native code within the same file. Thus, the compilation result includes two different versions of the source code at the same time.

additional semantic checks when objects are accessed. Java, for example, performs out-of-bounds checks for array accesses, or checks if pointers are not null before they are dereferenced [106]. In contrast, lower-level languages like C may directly access memory without checks.

When foreign language objects are accessed from C code in Sulong, Sulong forwards the access via Truffle’s interoperability framework to the corresponding language runtime of the foreign object [112]. This way, language-specific access checks are performed automatically even in polyglot applications, and language-specific semantics are preserved. Thus, if a Python or JavaScript object is accessed in Sulong, the access itself is done via the GraalPy or GraalVM-JavaScript runtime, respectively.

If one converted an object from a higher-level language like Python to native memory, the raw data would be accessible from native code. Checks by the higher-level language runtime would then be missing and the semantics of the higher-level language would be violated. Therefore, managed objects from a foreign language can neither grant direct access from native code nor can they be transformed to native memory automatically. In general, any managed object must be treated as a black box in native code.

4.3.2 Access Scenarios

In the following, we will show different access scenarios. We show what is necessary to access memory while keeping language semantics. Also, we explain why functions that access foreign objects have to be executed in managed mode, i.e., must not be executed natively.

To illustrate accesses from different execution modes to different memory representations, imagine a simple two-dimensional point data structure consisting of coordinates x and y as follows:

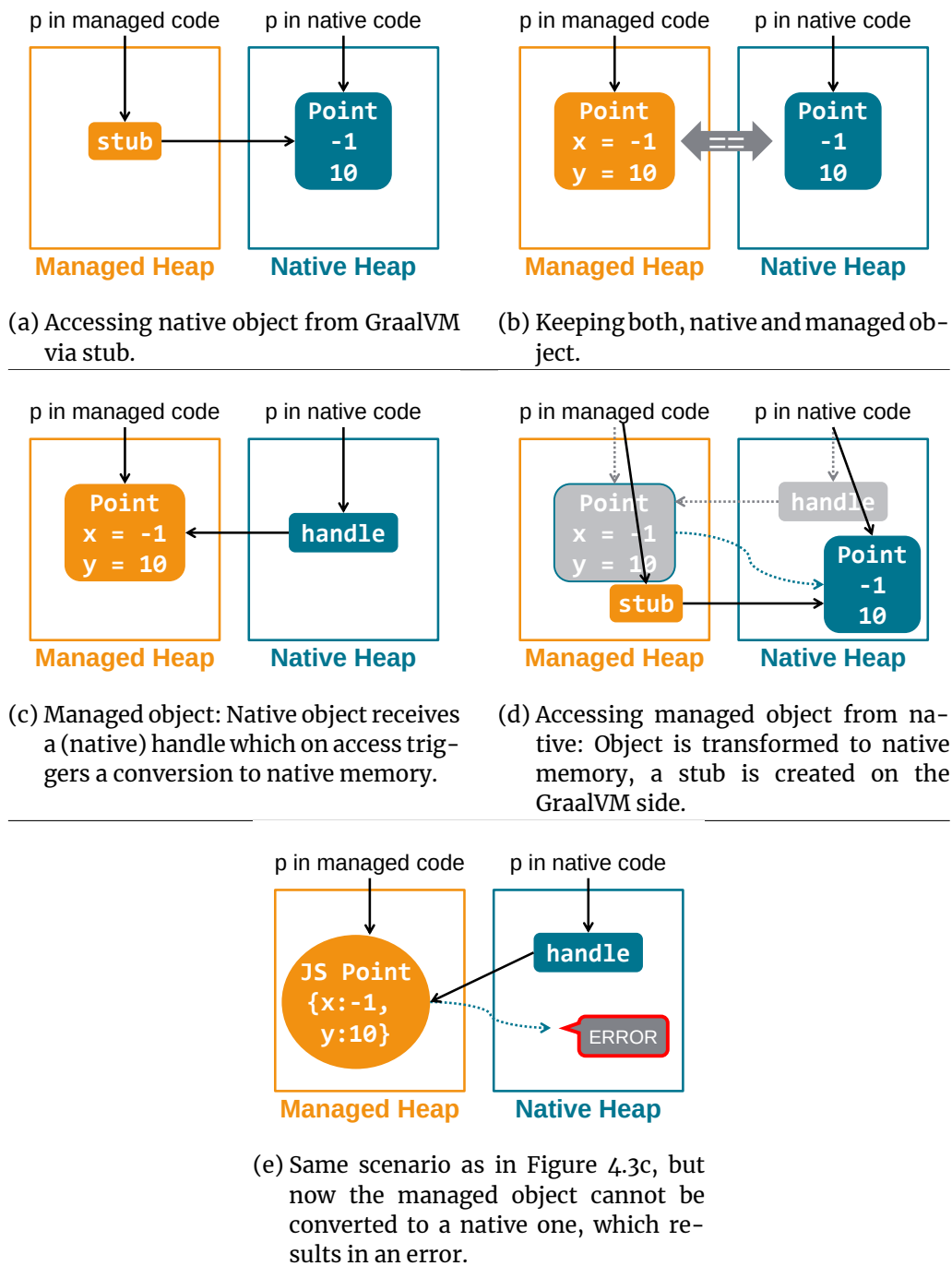


Figure 4.3: Memory object handling in different scenarios.

```
struct Point { int x; int y; }
```

We now create a concrete Point instance and store its pointer in a variable:

```
struct Point *p = (struct Point*) malloc(sizeof(struct Point));
```

Let us now consider different access scenarios of p in the following.

Native Heap

The content of *p* can be stored on the native heap. In this case, native execution is trivial. For access from GraalVM, a stub is created on the managed heap which forwards all operations to native memory, as shown in Figure 4.3a. The access from GraalVM to native memory also works via the Truffle NFI.

Duplication

The content of *p* can also be stored twice: Once in native memory and once on the Java heap, as shown in Figure 4.3b. Each execution mode accesses *p* in its own memory. However, each execution mode now accesses its own instance of the object. Thus, to avoid inconsistencies, duplication is only possible when the corresponding objects is marked as constant/read-only.

Managed Heap

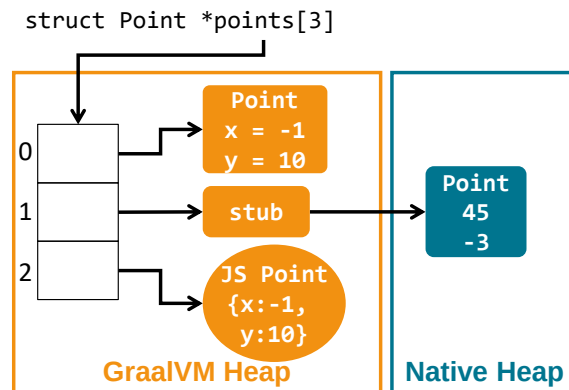
The remaining case is where the content of *p* is stored on the managed heap only, which is shown in Figure 4.3c: Native execution gets a handle to *p* that can be stored into variables and passed to other functions. As long as the pointer *p* is not accessed/dereferenced in native code, nothing has to be changed. However, when native execution accesses such a handle by dereferencing it, the managed object on the Java heap has to convert itself to a native memory object.

Function Pointer If the object is a function pointer, the function is executed in managed mode when invoked, and only the result is converted to native memory. This will be described in more detail below in Section 4.5.3.

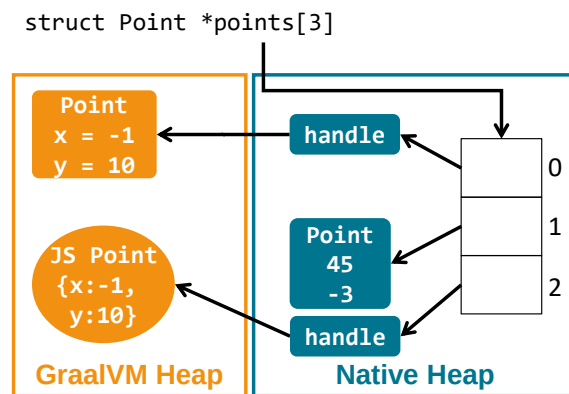
Conversion to Native Heap Possible If the conversion to native memory is possible for the corresponding managed object, as shown in Figure 4.3d, the situation is similar to Figure 4.3a, and program execution continues. This is the case for all managed objects that originate from C code, i.e., for all objects that have been created by Sulong and its managed LLVM execution. Therefore, the *Point* object in Figure 4.3c can also be transformed to native memory (given that it has been created by Sulong) as no language semantics are violated.

Conversion to Native Heap not Possible Otherwise, the object cannot be converted to native memory as shown in Figure 4.3e. This occurs when foreign language objects are accessed from native code. Thus, functions that access objects originating from a managed language cannot be run natively, but must be executed in managed mode.

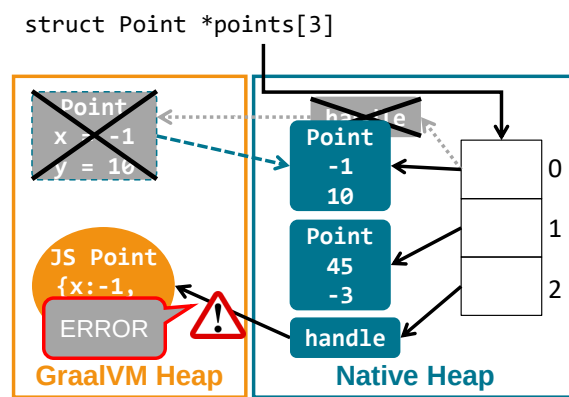
By default, the case where conversion to native memory is not possible would not happen very often, since managed execution via Sulong is preferred when both LLVM IR and native code are available. Therefore, most accesses to and from compiled C code (LLVM IR/native code) happen on the managed heap, which can deal with objects of foreign languages without any problems. However, using our hybrid execution mode, which increases the amount of natively



(a) C array on the managed heap, containing managed, native and foreign objects.



(b) C array on the native heap, elements not accessed from native execution yet.



(c) C array on the native heap, elements accessed from native execution. This works for managed and native, but not for foreign data.

Figure 4.4: Lazy conversion to native memory with a C array as an example: Managed array (Fig. 4.4a), array passed to native execution (Fig. 4.4b), and array elements being converted due to access of individual elements (Fig. 4.4c).

executed code, the scenario where natively executed code accesses managed data from other languages happens more often. Thus, we have to avoid this case.

A way of reducing the number of conversions from managed to native memory is to apply them lazily. As an example, imagine a managed C array containing pointers to elements, as in Figure 4.4a. `points[0]` contains a managed point object, `points[1]` a native point object, and `points[2]` a foreign point object. When the array reference is passed to native code, only the array itself gets converted to native memory as in Figure 4.4b. The references to the objects can still point to handles of the corresponding managed or even foreign objects. While `points[0]` and `points[1]` can be accessed safely in native code, the use of `points[2]` is not possible in native code, as in Figure 4.4c. Nevertheless, if `points[2]` is not accessed by native code, then this code can be executed natively.

Thus, if an array containing managed elements is passed to native code, the elements themselves are not converted to native memory if they are not accessed. This saves unnecessary conversions that might cause problems as described above.

4.4 Native Symbol Resolution

When mixing native and managed execution, it may happen that managed execution may access a symbol in native mode, e.g., may call a native function or may access a symbol stored in native memory. This cross-mode access can lead to several problems. These problems and our solutions for such cases will be explained in this section.

4.4.1 Retrieving Symbol Information

Information about native symbols, such as name or type, leads to an access of native code by default. However, to access such symbol information, it is not necessary to access native code: By using the Sulong toolchain as illustrated in Figure 4.2, the same C code is present as native as well as managed code. Thus, instead of performing a request for native symbol information, the existing data structure for the managed LLVM symbol information can be used, which is already implemented in Sulong.

4.4.2 Library Loading and Symbol Resolution

In a mixed mode environment, some symbols are only used in either mode (native/managed), but not both. Thus, it can happen that they are only resolved in either mode, too. If a native library now contains two symbols, where only one of them is needed, then an error can happen: By default, initializing the library also initializes all its (publicly accessible) symbols. However, unused symbols cannot be initialized.

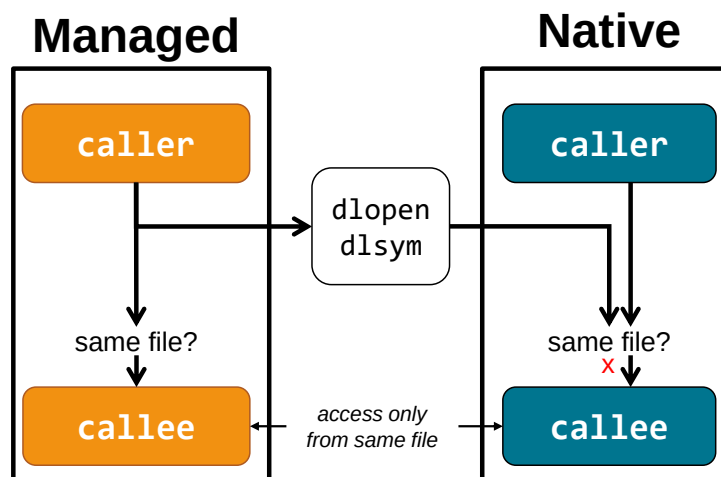


Figure 4.5: Symbol access within the same file. However, such an access within the same file, but from managed to native execution, looks like a foreign access from the perspective of `dlopen/dlsym`.

Thus, we modified the loading of libraries such that native symbols are resolved lazily. In other words, native symbols are only resolved at their first use, instead of when their library is loaded. This allows unused native symbols to remain unresolved, while used symbols are resolved correctly.

Related to this, also the scope of loaded native symbols has to be taken into account: In our approach, `dlopen` is used to load native libraries [57]. By default, symbols defined in a loaded library are not made available to resolve references in subsequently loaded libraries. This is a problem when different libraries access the same symbol.

Therefore, it is necessary to explicitly make symbols of a loaded library available for symbol resolution of subsequently loaded libraries. This can be done by using the `RTLD_GLOBAL` flag when loading a library with `dlopen` [57].

4.4.3 Resolution of Native Static Symbols

The `static` keyword in C keeps symbols local to a file. This means, static functions or fields are not accessible or callable from the outside, also not for the Truffle NFI, `dlopen` or `dlsym` [57, 58]. This leads to a problem if Sulong calls a static function of the same file in native mode. Both, `dlopen` and `dlsym` cannot detect that the access originates from the same source code file and deny straightforward access.

As an example, execution of `caller` in Figure 4.5 wants to get information about `callee`, which both originate from the same file. Such an access, denoted by the black arrows, is possible by default, as both symbols are defined in the same file. However, when mixing native and managed execution, it may happen that managed execution originating from a certain source code file may access a symbol from the same source code file, but in native mode. Such a lookup is performed via `dlopen` or `dlsym` [57, 58] and shown in Figure 4.5, too. For `dlopen`

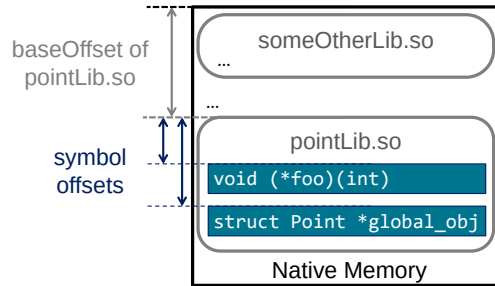


Figure 4.6: Calculation of the run-time address of a symbol in native memory.

and *dlsym*, such an access looks like a foreign access from another source file, and is thus blocked.

To retain access to the native symbol, we need to calculate its native address during run time, which consists of the following parts: The necessary values for this computation are illustrated in Figure 4.6. These necessary values for calculation include the relative symbol offset within the library, which is stored for every symbol in the symbol table of the library file.

- The symbol table of every native file contain the relative symbol offsets within the library.
- To determine the library’s base offset in memory at run time, a lookup call on the link map is conducted.

By summing these two numbers for each symbol, we can compute the pointers to native memory, even in situations where accessing symbol information is not possible for the Truffle NFI.

4.5 Switching Execution Mode: Method and Function Calls

The execution mode, native or managed, is specified for every function. Function calls constitute the boundaries between the two execution modes. Therefore, it is not possible to switch execution mode within a function. Managed execution is performed via Sulong, Truffle’s LLVM runtime as described in Section 2.6.3.

```

1  static struct Point *global_obj = init();
2
3  void fetch_point(struct Point *p) {
4      foo(p->x);
5      *p = *globalObj;
6  };
7
8  void foo(int par) {...};

```

Listing 4.1: C code containing two functions, where one is called from the other. The function *fetch_point* accesses a global variable.

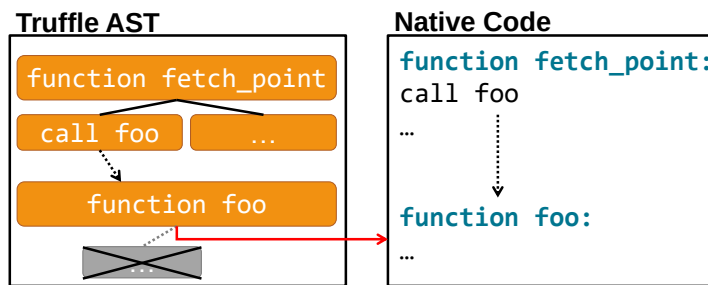


Figure 4.7: If `foo` should be executed native, then its call node in the LLVM Truffle AST has to be modified.

Listing 4.1 shows a simple function call, where `foo` is called from the `fetch_point` function. By default, the callee is executed in the same mode as the caller. In other words, if `fetch_point` is run on Sulong, in managed mode, then also `foo` is run on Sulong (when called from `fetch_point`). On the other hand, native execution of `fetch_point` would lead to a native execution of `foo`. Thus, every time when the execution mode should be changed, there is a modification necessary. The following subsections describe how we modified execution behavior to achieve such a change.

4.5.1 Calling Native Functions From Managed Code

When `fetch_point` is executed in managed mode, `foo` is also executed in managed mode by default. However, when `foo` should be executed in native code, a call from managed execution to native execution is needed. Such a call can be performed by the Truffle NFI: To switch execution mode in this case, we have to modify the LLVM Truffle AST, as shown with the red arrow in Figure 4.7. The call node in the AST does not contain the callee's AST as a child but gets a reference to the native function instead. This replacement, which ensures native execution of the callee, can be performed during the symbol initialization phase.

4.5.2 Native Execution: Limitations

Not every function can be executed in native mode when executing a polyglot application on GraalVM. The major reason why a C function must be executed in managed mode is that it accesses managed data which cannot be converted to native memory, as described in Section 4.3. For example, assume that a function `c` accesses managed data, as shown in Figure 4.8. This access requires `c` to be executed in managed mode, i.e., native execution of `c` is not possible.

By default, it is not possible to call managed functions from native callers, as native code is executed as is. In other words, native callers can only call native



(a) Execution model where native execution does not support calling managed functions: Managed execution of *c* implies managed execution of *a* and *b*. Thus, no hybrid execution in this case.

(b) Our execution model: Although function *c* is managed, *a* and *b* can be executed in either mode (native / managed).

Figure 4.8: Hybrid execution models for function *a* calling *b*, *b* calling *c*, and *c* accessing managed data. Dotted lines and gray nodes indicate that the call/method cannot be used.

callees. In the scenario of Figure 4.8a, this would lead to a major restriction: Managed execution of *c* would require all of its direct and indirect callers (*a* and *b* here) to be also executed in managed mode. Thus, such a hybrid execution model would be rather limited.

To open more opportunities for hybrid execution, it is necessary to allow managed functions to be also called from native callers. In the example of Figure 4.8b, *a* and *b* can also be executed in native mode, even if *c* is restricted to managed execution.

4.5.3 Calling Managed Functions From Native Code

It can happen that a natively executed caller invokes a callee that should run in managed mode, such as in Listing 4.2. To enable managed execution of callee, one would have to modify the native code.

```

1 void callee(void *ptr) {...} //should be run in managed mode
2
3 void caller(...) { //native
4     ...
5     //this way, bar_managed is still executed natively
6     callee(...);
7     ...
8 }
```

Listing 4.2: Problem: callee is still executed in native mode.

Simple, but not Feasible Workaround A relatively simple workaround to trigger managed execution of a callee (when invoked from a native caller) is shown in Listing 4.3: A global function pointer in the source code is initialized to a managed function in the LLVM Truffle AST. Its managed execution is even triggered if the function pointer is invoked from native execution. As a drawback, this

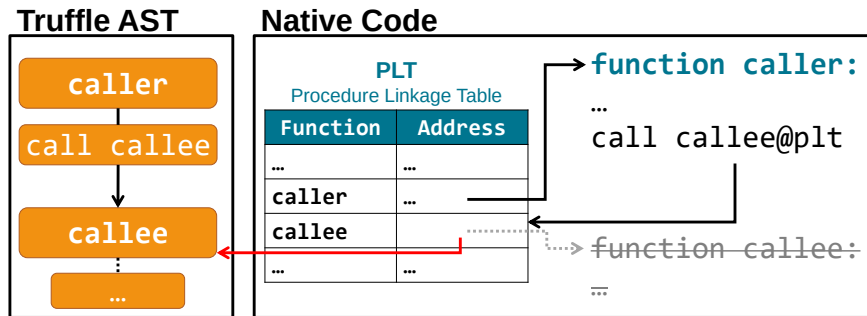


Figure 4.9: A change in the procedure linkage table (PLT) can make native callers call managed callees.

workaround implies a modification of the source code, as every function call has to be replaced by a call of the corresponding function pointer symbol. Additionally, the global function pointer has to be initialized in managed mode to point to the managed version of the function.

```

1 void (*callee_ptr)(void*);
2
3 void callee(void *ptr) {...} //should be run in managed mode
4
5 void caller(...) {//native
6     ...
7     //by calling the pointer, callee is executed in managed mode
8     callee_ptr(...);
9     ...
10 }
11
12 void init() { //has to be executed in managed mode!
13     callee_ptr = &callee;
14 }
```

Listing 4.3: Simple, but not feasible workaround to call managed functions from native code is to store every managed function in an explicit pointer variable. For that, `init` has to be executed in managed mode at the start of the program.

Using Lazy Linking When native code is linked and loaded dynamically, as it happens with the Truffle NFI, function calls cannot be resolved at compilation time. Instead, the compiler replaces every function call by a look-up into a procedure linkage table (PLT), as described in Section 2.4. This table is created and filled with placeholder values when native code is loaded. The first time a function is called, its address is resolved and stored into the PLT. For example, `caller` in Figure 4.9 calls `callee`, and a jump instruction to the resolved address of `callee` (shown by the dotted gray line) is written into the table. Subsequent calls to `callee` then only need a jump instruction to the table entry, which then invokes the corresponding callee.

This indirection can be used to trigger managed execution from native execution: Instead of resolving the address of the callee in native mode, we modified the Sulong runtime such that execution automatically resolves the address

of the same function in managed mode. Sulong then stores a jump to this address into the PLT, such that subsequent calls also invoke the callee in managed mode. This injection of the address of the function in managed mode is indicated by the red arrow in Figure 4.9.

In contrast to changes within GraalVM, this modification of native code is platform-dependent. Therefore, we implemented this patching of the PLT only for code on x86-64 Linux machines as a proof of concept. Other architectures require a similar procedure.

4.6 Foreign Caller Calls Native Callee

In our context, managed functions are C functions executed on Sulong. However, there might be other managed functions (e.g., Python functions) that are not executed on Sulong. As introduced above in Section 4.1, we call such functions *foreign* functions. It can happen that a C function c is called by such a foreign function b_{foreign} and that our automated heuristics recommend to run c in native mode, as shown in Figure 4.10a. However, it is not possible to simply call a native C function directly from a foreign language, as the Sulong runtime performs extra work such as adding context parameters.

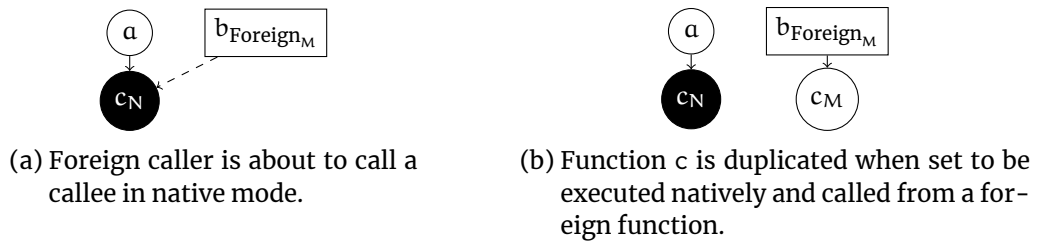


Figure 4.10: Functions may get duplicated — Call structure (Figure 4.10a) and a possible execution scenario (Figure 4.10b).

In such a situation, we create a duplicate version of c on demand: If invoked by a managed caller a via Sulong, c runs in native mode as recommended. If invoked by a foreign caller b_{Foreign} (not via Sulong), the managed version of c is called, as depicted in Figure 4.10b.

Unlike the situation where managed data is accessed from native execution, detecting when a native function is called from a foreign function can be done at run time. This means that there is no need to detect such calls ahead of time. Instead, the callee's code can be dynamically replaced at run time if an error occurs during the method call.

4.7 Limitations

Similar to many optimizations, the approach presented in this chapter may not always lead to a better result. An important limitation concerning the implementation of the system is the risk of errors at run time when functions in native mode access managed data that cannot be converted to native memory.

While such failures cannot be detected upfront, our system lacks the ability to rewind native execution upon encountering a memory error, forcing the problematic code to resume under managed execution.

4.8 Summary

In this chapter, we presented a hybrid execution environment for native extensions. Based on Truffle, which allows seamlessly mixing code of different languages, we propose to combine managed and native execution for the code of native extensions. While calling native code from a managed environment is straight-forward, calling into Sulong from native code requires patching. Similar to the implementation of the approach in Chapter 3, we achieved this by using data structures for lazy linking, concretely PLT and GOT.

Accessing heap objects, however, turned out not to be trivial, as native and managed code use different heaps as well. Therefore, it is necessary to create access stubs in managed mode to enable access to native memory. For the other direction, it is necessary to convert managed objects to native memory upon access. We chose lazy access of objects in the managed heap, as this prevents conversions to native memory that are not necessary. However, native execution cannot access foreign objects, as those objects cannot be transferred to native memory without changing their specific language semantics. Therefore, code accessing foreign objects has to be executed in managed mode.

Chapter 5

Heuristics for Mixing Execution Modes in Native Extensions

In Chapter 4, we proposed hybrid execution for native extensions. In this chapter, we strive for an automated approach of such a hybrid execution, where heuristics suggest an execution mode for every function automatically. We describe features of functions that advise us to select a specific execution mode for certain functions. For the remaining functions, we propose a call graph based analysis. We also show that heuristics can enable performance optimizations that have been missed by manual selection.

A short terminology recap of Chapter 4: We call C code to be executed in *native* mode when it is AOT compiled and executed natively. *Managed* execution denotes execution via Sulong, i.e., GraalVM's LLVM runtime with interpretation and JIT compilation. *Foreign* code belongs to another language that uses the C code as a native extension, such as Python or JavaScript. Similarly, the term of *foreign* objects belongs to objects passed from foreign code to code of the native extension.

5.1 Motivation

An example for a situation where an automated approach is desired can be an application using the Python lxml package [56], which is an XML parser written in C. Applying our approach of Chapter 4, which is mixing native and managed execution for the C code, to lxml leads to a call graph of more than 8000 nodes. It is not feasible for us to consider the high amount of executions manually: Assuming n of these functions access managed data and thus have to be executed in managed mode, there are still $8000 - n$ functions left where both execution modes are possible. This leads to 2^{8000-n} possibilities how to combine different execution modes. Furthermore, even if manual annotations were feasible, one might easily miss optimization opportunities.

Problem With Determining the Execution Mode Staticallly For every function in the native source code, we have to decide if the managed runtime should choose to interpret and JIT-compile it, or if the AOT-compiled code should be called. Thus, we have to provide an oracle that takes a function name as an input, and determines which execution mode (native or managed) to run.

To achieve a fast warm-up, it would be desirable to execute as much code as possible in native mode. However, as discussed in detail in Section 4.3, there

are scenarios where native execution is not possible, such as when native code accesses foreign objects on the managed heap.

A solution for that problem would be to apply taint tracking, such that all types that might contain foreign data are known. Such types can then indicate that execution should be done in managed mode, for example. Unfortunately, it cannot always be determined statically if a function can be executed natively. This is because every pointer in C and C++ code can possibly point to a managed object. For a given C array of pointers to objects, such as the example described in Figure 4.4 in Section 4.3.2, it is impossible to statically detect whether accessing certain elements has to happen in managed mode. Thus, we want to apply heuristics based on dynamic profiling data that select the execution mode based on an estimation of execution time.

5.2 Function Execution Time in Different Execution Modes

To determine the execution mode of a function, this section is about execution times of function calls in different execution modes. Our goal is to find correlations between a feature and the performance of functions. This allows us to derive heuristics, as we describe in Section 5.3 below, from the findings of this section.

To obtain data of functions, we added a simple instrumentation to the Sulong runtime and perform a run of the program under Sulong in pure managed mode. Thereby, every call to a function and every access to a variable is tracked. With that data, we can build an extended call graph of parts of the executed application. Every node represents a function or a variable, while a directed edge represents a function call or an access to a variable. In contrast to a call tree, where one function is represented as multiple nodes if it is called from more than one call site, we decided to build a call graph. Thus, every function only has one corresponding node in the graph.

The following subsections show how such a call graph can be used to design heuristics for hybrid execution in order to improve the performance of a program.

5.2.1 Execution Time Estimation for Single Functions

For determining the execution mode for each function, the execution times of functions in managed mode and in native mode are important. For that, we estimate it based on static code size and control flow information, which is what the following subsections are about.

Execution Time Estimation Without Control Flow

In general, the execution time of a function depends on the number and time of executed instructions. Ignoring control flow (loops and branches) initially,

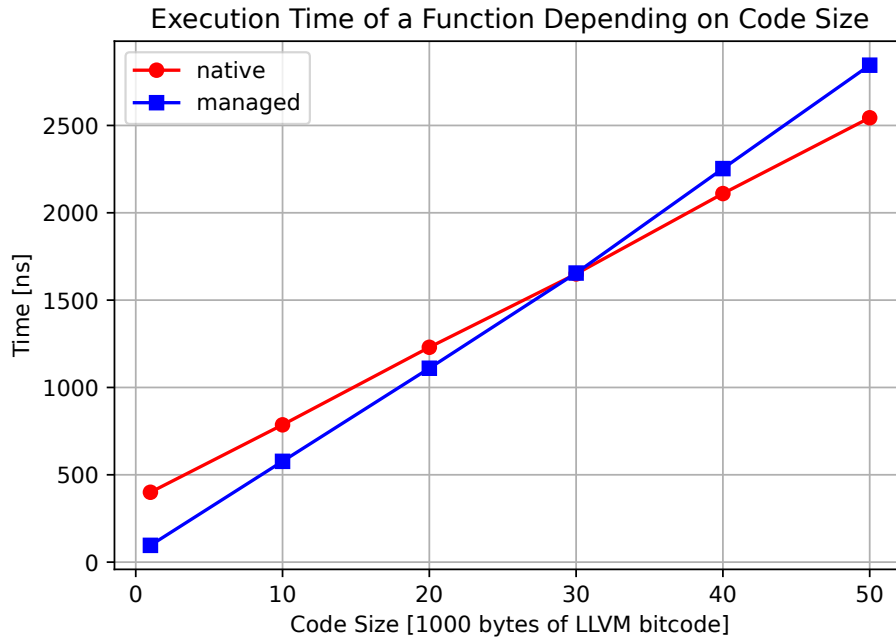


Figure 5.1: Execution time of a function called from Sulong without branches and loops, based on its code size of LLVM bytecode.

there is a positive correlation between the code size of a function and its execution time.

We therefore performed measurements with functions of different sizes that do not contain loops or branches. The functions are taken from the gzip, nbody and mandelbrot benchmarks. We performed those measurements on a system with an Intel Core i7-1255U CPU, 32GB RAM and Ubuntu 22.04. During measurements, turbo boost and network connections were turned off. We used GraalVM 24. Concerning language versions, we used Java 23, Python 3.10.8 and LLVM 16. Each benchmark was run 10 times for each of the settings. Per function, we profiled the self time spent in this function with Truffle’s CPU sampler [35, 71] and divided it by its call count. The call counts reached from 29316 to 914952. With high call counts instead of single call measurements, we account for system noise and profiling inaccuracies.

The measurements consist of micro-benchmarks, where functions with different sizes of LLVM bytecode are called from Sulong. This also means that the measurements for native execution of the called function include the call overhead for the call from Sulong to native execution. Although this overhead will be considered in more detail in Section 5.2.2, we decided to include it in these measurements as well: Setting a function to native mode will still imply a switch from managed to native execution, introducing time to switch execution mode.

The results are shown in Figure 5.1, which show a break-even point at a code size of ≈ 30 kB. Therefore, we suggest native execution for functions with an LLVM bytecode size larger than 30 kB.

```
1 while(...) {//1
2   while(...) {/ * 2 */
3   while(...) {
4     ...//3
5     if(...) { ... }
6     ...//7
7   }
8 }//8
```

Listing 5.1: Sample code containing loops.

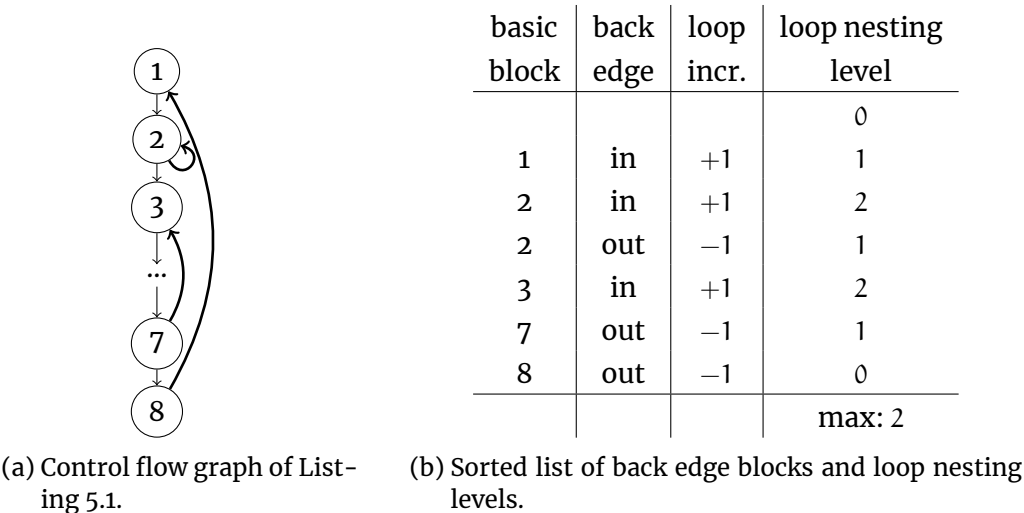


Figure 5.2: The maximum loop nesting level of Listing 5.1 is calculated.

Control Flow Considerations

As mentioned above, control flow can heavily influence the execution time of a function: While branches (e.g. `if` instructions) skip parts of the code during execution, loops lead to multiple execution of the same code. Therefore, the proposed heuristics of taking 30 kB of code size as a threshold is too inaccurate. Branches affect the ratio of executed code by less than 50 % on average, which is an amount we ignore for our heuristics. However, loops that are executed multiple times can change the amount of executed instructions by more than one order of magnitude. Therefore, it makes sense include information about loops into our heuristics.

As our application code is LLVM bytecode, which is structured as a linearized control flow graph with basic blocks [50], arbitrary control flow is supported. Thus, it is not trivial to detect loops and the maximum loop nesting level out of the box. Therefore, we look at back edges in the control flow graph of the LLVM bytecode, and assume every back edge to indicate a loop. Back edges can be found easily, as every basic block in LLVM is annotated with an ascending block index. For example, Listing 5.1 shows source code containing loops. The corresponding control flow graph is shown in Figure 5.2a, where back edges are shown in bold.

When parsing the bytecode, we traverse the control flow graph and check every jump between basic blocks. If the block index of the target block is smaller

than the index of the current block, we have found a back edge. Each block with an incoming or outgoing back edge is stored in a sorted list, as shown in the left column of Figure 5.2b. This list is then traversed from top to bottom. Every incoming back edge means that a loop is entered, so we increment the loop nesting level. Conversely, every outgoing back edge decrements the loop nesting level, as shown in the right column of Figure 5.2b. To consider loops within the same node, it is important that incoming edges are sorted in front of outgoing edges of the same node, as it happens for control node 2 in Figure 5.2. The maximum number during traversing this list is therefore the maximum loop nesting level of the corresponding code.

For a precise calculation of the execution time, one would need the code size inside the loop as well as the number of loop iterations during execution, which depends on input parameters of the program. Therefore, the numbers for loop iterations cannot be known by static code analysis.

Since we only deal with heuristics, we simplify our approach and assume that each loop level causes the function to run 10 times as long as without loops. Although we are aware that this factor 10 varies for different programs and even varies for different loop levels within one program, assuming 10 as a multiplication factor for each level turned out to be a reasonable choice.

Therefore, our heuristics which sets large methods to native execution changes from $s(f) > 30 \text{ kB}$ to $s(f) \cdot 10^{\text{loopDim}} > 30 \text{ kB}$, where loopDim denotes the maximum loop nesting level. For example, if the function in Figure 5.2 had an LLVM bytecode size of 400 B, the size would be estimated to be $400 \text{ B} \cdot 10^2 = 40 \text{ kB}$. While the size of f itself (400 B) would be too small to trigger native execution, the loops in f make it large enough to suggest native execution.

Effect on Warm-Up Performance

Although looking on code size rather than investigating how single functions affect warm-up performance, we assume larger functions to cause a higher compilation effort. By setting functions larger than 30 kB to native execution, we reduce the effort for JIT compilation and thus expect a faster warm-up.

5.2.2 Function Call Overhead of a Single Function

Although the execution time of a function, as described in Section 5.2.1, is important, the costs of the function call overhead also have to be taken into account. Especially when a managed function calls a native function and vice versa, context information of the managed runtime has to be added or removed, respectively. Thus, we also ran micro benchmarks which measure the function call overhead within and across execution modes. We tracked the time directly before and after method calls. We performed those measurements on a system with an Intel Core i7-1255U CPU, 32GB RAM and Ubuntu 22.04. During measurements, turbo boost and network connections were turned off. We used GraalVM 24. Concerning language versions, we used Java 23, Python 3.10.8 and LLVM 16. Each benchmark was run 100 times for each of the settings. The results, depending on the execution mode of the caller and the callee, are shown in Figure 5.3.

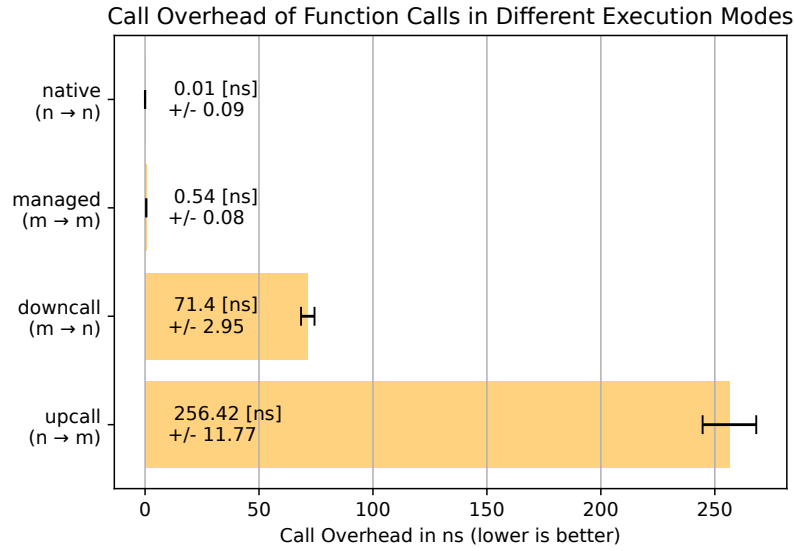


Figure 5.3: Function call overhead in different execution mode scenarios.

When both the caller and the callee are executed in the same execution mode, a function call typically involves not more than a couple of machine instructions. Thus, the time for calling and returning from a method can be neglected in this case.

However, a downcall, which is calling a native function from a managed caller, significantly increases the call and return overhead by two orders of magnitude compared to purely managed execution. An even higher overhead is imposed by an upcall, which is when a native function calls a managed callee. Therefore, it is necessary to take a look at function call overheads when assigning execution modes to functions. Otherwise, the worst case could be a scenario where execution switches between native and managed mode back and forth, causing high overhead costs.

As a first step, we take a look at the call overhead for every function f that has no hard constraints of being executed in native or managed mode. Call overheads happen due to callers and callees of f , where both (callers and callees) can be executed in native or in managed mode. Combining these possibilities leads to four different cases: We can distinguish between managed and native functions on the one hand, and callers and callees of f on the other hand, as shown in Figure 5.4. The edge weights (in_m, out_m, in_n, out_n) denote the absolute frequencies of the calls.

If f is executed in native mode, we get the following costs, where d and u denote the costs of a downcall and upcall, respectively: Managed callers of f perform a downcall, thus the costs are $in_m \cdot d$. Managed callees of f cause an upcall, thus the costs are $out_m \cdot u$. Native callers and native callees hardly cause call overheads as shown in Figure 5.3 and are neglected. For callers and callees that have no execution mode assigned yet, we assume that half of the calls will be native and half will be managed. Therefore, the amounts of managed calls are increased by half of the amount of the undetermined calls (named as in_u and out_u). Thus, the call overhead costs when f is native can be computed as in Equation 5.1.

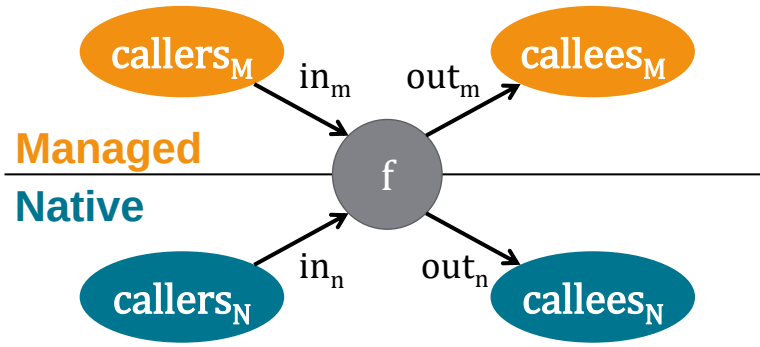


Figure 5.4: Function f can be executed in managed mode as well as in native mode. Since overhead costs for upcalls and downcalls of f also influence execution time, they are taken into account.

$$t_{\text{call}}(f, n) = \left(\text{in}_m + \frac{\text{in}_u}{2} \right) \cdot d + \left(\text{out}_m + \frac{\text{out}_u}{2} \right) \cdot u \quad (5.1)$$

If f is executed in managed mode, we get the following costs: Native callers of f perform an upcall ($\text{in}_n \cdot u$), whereas native callees of f cause a downcall ($\text{out}_n \cdot d$). Managed callers and callees can be neglected for our heuristics. Half of the undetermined calls are added to the native calls this time. Thus, the call overhead costs when f is managed can be computed as in Equation 5.2.

$$t_{\text{call}}(f, m) = \left(\text{in}_n + \frac{\text{in}_u}{2} \right) \cdot u + \left(\text{out}_n + \frac{\text{out}_u}{2} \right) \cdot d \quad (5.2)$$

5.3 Heuristics

From the above observations, we design the following heuristics for a hybrid execution mode: First, we mark all functions that access managed data for managed execution. This allows us to avoid an invalid access from native code as described in Section 4.3. Then, we mark all functions that are large enough for native execution. Finally, we traverse the remaining graph and set the execution mode of the remaining functions such that the call overhead is as low as possible.

For that traversal, we go through the graph via a breadth-first search (BFS) and thus use a queue as a data structure. We prefer BFS over depth-first search (DFS), since we have multiple starting points, and since nodes in the call graph affect all of their neighbors.

In detail, the heuristics look as follows: Given F as the set of all C functions and F_M and F_N as the set of all C functions in managed and native mode respectively, heuristics for an automated hybrid mode can be seen as a partitioning function $F \rightarrow \{F_M, F_N\}$. The heuristics work as follows:

1. Mark all C functions with managed parameter types or managed return types for managed execution.

2. For each remaining C function f , check if f is large enough for native execution, as shown in Section 5.2.1. I.e., if $s(f) \cdot 10^{\text{loopDim}(f)} > 30 \text{ kB}$, mark f for native execution.
3. Add all neighbors (callers and callees) of functions marked as managed (step 1) or native (step 2) to a queue Q . While Q is not empty:
 - a) Remove a function from Q and calculate $t_{\text{call}}(f, n)$ and $t_{\text{call}}(f, m)$ as in Equations 5.1 and 5.2.
 - b) If $t_{\text{call}}(f, m) > t_{\text{call}}(f, n)$, i.e., if the call overhead is higher for managed execution, then set f to native execution. Otherwise, mark f for managed execution.
 - c) Add all neighbors (i.e., callers and callees) of f without an assigned mode to Q .

5.4 Evaluation

We evaluate the proposed heuristics by assessing their impact on the performance on `lxml` [56] and `ujson` [102, 103] benchmarks. Although Python comes with a standard performance benchmark suite [99], we did not select these benchmarks, as our hybrid execution mode operates on C source code and requires native extensions. Therefore, we aim for a scenario where the primary application is written in a high-level language such as Python but uses native extensions written in C, such as an XML or JSON parser (`lxml` and `ujson`, respectively). We selected these benchmark suites since the actual computation of the benchmark happens in C code. Python code acts as a wrapper around the native extension computation. While managed objects are indeed accessed by C/C++ functions, which thus have to be executed in managed mode, there are also C/C++ library functions that can run in native mode.

5.4.1 `lxml` Benchmarks

Benchmark in General As a first benchmark suite, we decided to use the `LXML` Python package for parsing XML data to objects [56]. This python package also comes with a benchmark harness. While the main program of the benchmark is written in Python, the tasks themselves are written as native extensions, which is what we are interested in.

The harness first parses and allocates XML data structures in memory during its setup process. Then over several iterations, operations are performed on the parsed XML trees: `annotate` annotates nodes of XML tree, while `path` performs path searches within the XML tree.

These tasks (setup, annotating, path search) are performed on different trees: There are four basic tree shapes (trees with many grandchildren, wide but flat trees, narrow but deep trees, small trees). Each of those four differently-shaped trees then again exists in three versions: No text encoding, string as text type and unicode as text type. Optionally, trees can be attributed.

Methodology In general, we used fully managed execution on Sulong as the baseline and compared it to the execution times of our automated hybrid execution mode, as well as to a mode where all functions are native that do not access managed data.

We performed the measurements on a system with an Intel Core i7-1255U CPU, 32GB RAM and Ubuntu 22.04. During measurements, turbo boost and network connections were turned off. We used GraalVM 24. Concerning language versions, we used Java 23, Python 3.10.8 and LLVM 16. Each benchmark was run 10 times for each of the settings, with 20 iterations of the core benchmark. Thus, per benchmark and per configuration, we collected 200 data points to account for system noise.

We used lxml version 2.9.13. For data setup, each of the four available trees T1 to T4 is parsed in six different versions (attributed/not attributed, combined with either of the three options <no text/string/unicode>), leading to $4 \cdot 2 \cdot 3 = 24$ different data points. For annotation, the non-attributed trees are used, leading to twelve different scenarios (4 trees, combined with either of the three options <no text/string/unicode>). For path search, non-attributed trees without text are used, leading to four different scenarios (4 trees). Although lxml consists of microbenchmarks, it has been designed to consist of different tasks to represent different kinds of workloads [90]. We measure the time the data setup takes, as well as warm-up and peak performance of the annotation as well as the path search task.

Questions To evaluate our automated hybrid execution mode, we are interested in the following questions:

1. How does our hybrid execution model perform with respect to the particular allocation of objects and variables? How does performance differ between allocation in native and managed memory?
2. How does our hybrid execution model perform with respect to the warm-up performance of the different benchmark tasks?
3. How does our hybrid execution model perform with respect to the peak performance of the different benchmark tasks?

For question 1, we measured the total data setup time. For question 2, we measure the warm-up performance of both, the annotation and path search task. We consider warm-up to be finished as soon as execution time is within 5% deviation from the maximum observed value. For question 3, we measure the peak performance of both, the annotation and path search task. We consider peak performance as the median value after warm-up.

Results Let us first look at the result of the heuristic: Concerning global variables, all C variables are stored in native memory, while 22 % (804 out of 3637) of all C functions are selected for native execution.

The results of the three measurement questions from above are shown in Figure 5.5, where the speedup of our hybrid execution mode is shown in blue. The results where all possible candidates for native execution are executed in native mode are shown in red. Given that the end of the warm-up phase of VMs is not

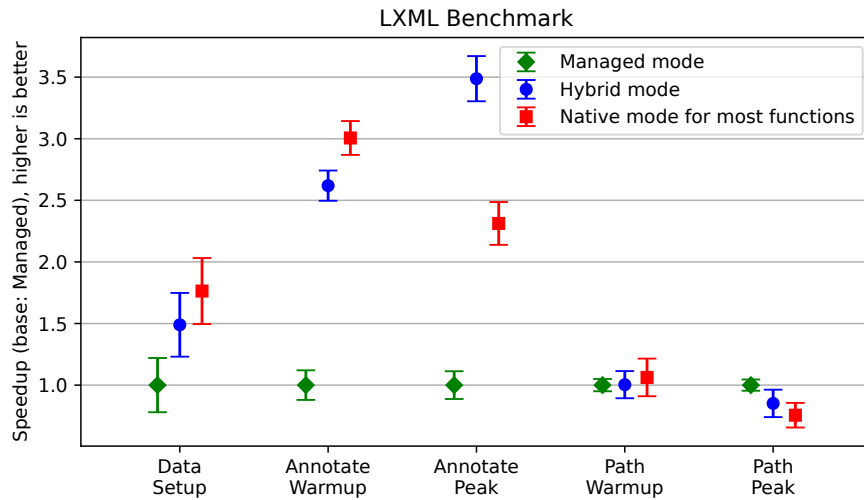


Figure 5.5: Results of LXML Objectify Benchmark: For data setup and warm-up performance, native execution performs better. For annotation tasks, our hybrid mode pays off. For path search within the tree, managed execution performs best.

always easy to determine [7], we see that similar benchmarks using the GraalPy runtime need about five to ten iterations to warm up, which corresponds to our measurements.

Hybrid execution leads to an average speedup of $1.48\times$ for the setup time of the given benchmark on average, native execution achieves an average speedup of $1.77\times$. However, the standard deviation is quite high, indicated by the error bars. Nevertheless, we see an advantage of hybrid execution mode compared to managed execution for question 1.

Concerning warm-up and peak performance of the LXML benchmark, the answers to questions 2 and 3 highly depend on the kind of task: Annotation tasks on the XML tree achieve an average speedup of $2.6\times$ for warm-up performance. For peak, hybrid execution achieves a speedup of $3.5\times$, which is even higher than with native execution. Such a behavior can also be seen at the *ujson* benchmark in Section 5.4.2. Compared to the baseline (all functions executed in managed mode on Sulong), executing some functions in native mode clearly pays off. Running as many functions in native mode as possible implies even less optimization work for the JIT compiler, which leads to a slightly higher speedup for warm-up, but then results in a slower peak performance.

For the tasks that aim to find certain paths within the XML tree, the situation is different: While the warm-up performance in hybrid mode does not show a significant difference to managed execution, there is a small drop in peak performance with a slowdown of $0.85\times$, and an even worse drop for running as many functions natively as possible. In this case, native execution of certain functions did not speed up the execution process. A possible reason is that the GraalVM JIT compiler is able to optimize the managed functions effectively, which surpasses advantages of native execution. As well, managed execution may enable the GraalVM JIT compiler to inline managed data accesses and to optimize them efficiently. The results indicate that the performance of hybrid

execution highly depends on the executed code, as well as on the data dependencies.

We will discuss the results, especially similarities to the ujson benchmark in Section 5.4.2, in more detail in Section 5.5.

5.4.2 UltraJSON Benchmarks

The basic principle behind how our hybrid execution mode can achieve speedups can be seen with the benchmark available in the GitHub repository of the Python UltraJSON package [102, 103]. The package itself offers efficient loading, parsing and dumping JavaScript object notation (JSON) data.

Benchmark in General The benchmark loads and parses JSON data as a first task, which is then dumped again in a second task. There are nine different JSON files that consist of different data structures: 5 different arrays with 256 elements each (string-int dictionaries, doubles, strings, boolean values, UTF-8 strings), one medium complex object, one complex object, two different dictionaries with 256 arrays each (one of them outputting sorted keys).

Methodology Again, we used fully managed execution on Sulong as the baseline and compared it to the execution times of our automated hybrid execution mode, as well as to a mode where all functions are native that do not access managed data. Thus, also similar to lxml, we ran this benchmark in three modes: Fully managed (1), our hybrid execution mode (2), and a mode where as many functions as possible are executed in native mode (3).

We performed the measurements on a system with an Intel Core i7-1255U CPU, 32GB RAM and Ubuntu 22.04. During measurements, turbo boost and network connections were turned off. We used GraalVM 24. Concerning language versions, we used Java 23, Python 3.10.8 and LLVM 16. Each benchmark was run 10 times for each of the settings, with 50 iterations of the core benchmark. Thus, per benchmark and per configuration, we collected 500 data points to account for system noise.

For ujson, the used version was 5.10.0. We measure peak performance and look at the performance behavior of the parsing and dumping phases. The time of both tasks is measured within 10 fork runs, where each performs 500 iterations of loading and dumping.

Question To investigate behavior of different code within our automated hybrid execution mode, we are interested in the following question:

- How does our hybrid execution model perform with respect to the different phases of the ujson benchmark, i.e., parsing and dumping data? Is native or managed execution clearly better in either or both of these tasks?

For this question, we measure the peak performance of both, the annotation and path search task. Again, we consider peak performance as the median value after warm-up.

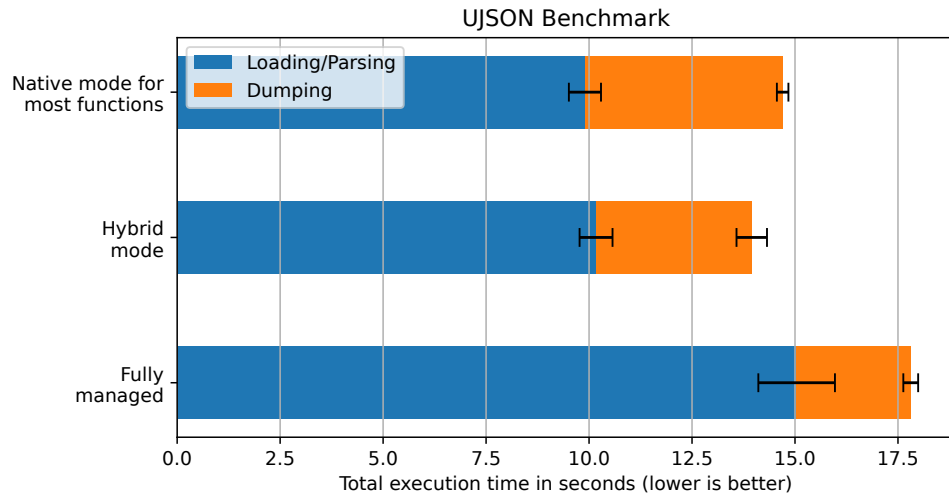


Figure 5.6: Results of the UJSON Benchmark: Our automated heuristics for hybrid execution achieve an overall speedup.

Results Our heuristics, as described in Section 5.3, proposed 76 % (725 out of 949) of the functions to run in native mode.

The results to our question are shown in Figure 5.6. While managed execution on Sulong is faster for dumping, (mostly) native execution is faster for loading and parsing. Concerning execution times, our hybrid mode, which combines native and managed execution, is between managed and native execution. In general, matches with results of the lxml benchmark in Section 5.4.1, where parsing data was also faster in native mode than in managed mode. For other tasks, such as traversing the tree, managed execution was faster.

However, it is interesting to look at the total accumulated execution times of both subtasks: In this case, for the whole benchmark application, our suggested hybrid execution mode strikes a balance and is faster than fully managed or close-to-native execution, which can be seen in Figure 5.6. While it is hard to generalize from two benchmark suites to the whole world of applications, it is a proof of concept that automated heuristics can achieve better results than a manual selection of the execution mode.

5.5 Discussion

Both benchmarks (Sections 5.4.1 and 5.4.2) show that parsing is faster in native code, whereas traversing data structures (for dumping or searching) is faster in managed mode.

A hybrid execution mode combines these strengths and can yield overall speedups beyond either mode alone. As demonstrated in the ujson benchmarks (Figure 5.6), we found a case where automated hybrid execution outperforms manual mode selection by the programmer. This holds regardless of whether the programmer uses explicit annotations or relies on the traditional split where Python runs managed and C extensions run natively. We therefore recommend performing parsing natively, while traversals, especially over potentially managed data, should run in managed mode.

One plausible explanation is that the GraalVM JIT compiler optimizes managed functions effectively enough to outweigh the benefits of native execution. Managed execution may also enable the JIT to inline and optimize accesses to managed and even foreign data. Overall, the performance of hybrid execution depends strongly on both the executed code and its data dependencies.

As with many optimizations, the approach in this chapter does not always improve performance. Speedups for a function depend on its implementation: Some functions run faster in native mode, while others are faster when they are interpreted and JIT-compiled. As we have shown in Section 5.4, this variation appears even within a single benchmark. The `ujson` case in Section 5.4.2 further shows that hybrid execution may not improve every part of an application. Nevertheless, it serves as a proof of concept that an automated decision can outperform manual selections.

5.6 Summary

In this chapter, we presented heuristics to determine which functions of a native extension should run in managed mode, and which should run in native mode. The evaluation shows that integrating an automated hybrid execution mode into GraalVM can improve the performance of mixed-mode programs.

Function size combined with loop nesting level information is a simple, but adequate heuristic for determining the execution mode. In contrast to the system we described in Chapter 3, calls between different execution modes take a significant amount of time. Therefore, we proposed heuristics that take these cross-boundary calls into account as well. However, it turned out that our heuristics do not always lead to a performance improvement over both baselines (100% managed vs. mostly native). Still, we see potential in popular Python packages such as `numpy` or `pytorch`, which both use native C extensions for performance improvement. Additionally, we showed that we can overtake performance of a manual execution mode selection by applying an automated heuristic.

Chapter 6

Related Work

The field of optimizing runtime systems has developed various approaches to balance trade-offs. First, we will dive into technical concepts that try to achieve such optimizations. Then, we will discuss related heuristics.

6.1 Technical Concepts

We first discuss technical concepts that share goals with our approaches, i.e., concepts that balance trade-offs in modern runtime systems. Mostly, the trade-off is about performance on the one axis, and resource utilization on the other axis, such as reducing file sizes or compilation effort while keeping peak performance high.

Copy and Patch Compilation *Copy and patch* is a recent compilation approach with low compilation time and high peak performance [116]. Its core idea is to match pre-compiled machine code templates to parts of the IR, such as an AST, and patch parameters such as constants or memory addresses afterwards. Although copy and patch compilation is orders of magnitude faster than existing compilers, the produced machine code can approach the peak performance of traditional compilers [116]. Since 2024, copy and patch compilation is also part of the official CPython runtime [88].

However, it does still not solve the problem of large binaries or offering hybrid execution for native extensions. Nevertheless, it can be considered as an alternative to classical AOT compilers, as it comes with a lower compilation time.

Project Crema Project crema [82] is a GraalVM project under current development. Its concept is to cancel the closed world scenario, i.e., to enable class loading at run time. Such loaded code is interpreted and optionally JIT compiled.

While the goal and approach is the same as ours in Chapter 3, their implementation is not ready yet concerning heuristics. However, our key findings of Chapter 3, such as the high memory and low run-time impact of static initializers in Java, are planned to be included into their heuristics in production. This will be discussed in more detail in Section 7.1.

6.2 Compiler Heuristics

Compiler heuristics play a crucial role in making selection processes within compilers and runtimes. The earliest compiler heuristics have probably been proposed by Backus et al. [5]. Simply, but effective, they assumed loops to be executed more often and applied optimizations such as loop-invariant code motion. While the heuristics of their work strongly developed and have made their way to sophisticated heuristics used today, they nevertheless helped to establish the concept of automatic programming and compilation. In the following, we describe heuristics that are related to ours.

6.2.1 Adaptive and Dynamic Compiler Heuristics

The concept of a JIT compiler is based on run-time information, which is then further used for dynamic compilation. Thus, JIT compilers are bound to rely on heuristics, where we would like to revisit the following existing approaches.

Mosaner et al. [62] propose a novel approach for dynamic compilers where machine learning models replace hand-crafted heuristics and are updated at run time. The system collects performance data during program execution and uses it to refine the model, allowing the compiler's optimization decisions to adapt to the specific environment. This is an application of machine learning to compiler heuristics that creates a self-optimizing system. Compared to static heuristics, the performance results of their approach reach from a 7% slowdown to a 30% speedup. However, our approaches in this thesis require the decision of the execution mode at compile time already. Also in the case of subsequent runs, data from previous compilations are hard to reuse in this case.

The approach of reusing code from previous compilations is nevertheless relevant: Mehta et al. [59] explore the reuse of code compiled by a JIT compiler across different program runs to reduce warmup costs. Subsequent application runs only need 11% of the original JIT compilation time. The relevance to heuristics lies in the proposed mechanism for selecting which previously compiled function to use. Their proposed mechanism to select which previously compiled function to use uses *speculative contextual dispatch*. This allows choosing a version of a function from a persistent code repository, which is indexed by the context and interpreter information in which the function was originally compiled. A similar concept is also applied for Project Leyden [83], available since Java 25, where JIT-compiled code is cached for future runs. The selection process is a heuristic that predicts which cached, pre-compiled code will be most effective for the current execution context to avoid redundant compilation. For that, they compare and order the different contexts based on inferred types and their hierarchies. As an example, a function receiving a sum type argument of `(int | String)` can be used if the actual current argument is of type `int`. However, in our hybrid execution context, we do not deal with code optimizations based on speculations and assumptions: When we have to select between execution modes at compile time, we do not have run-time contexts, especially run-time type information available.

While Pečimúth et al. [74] try to achieve the same goal as Mehta et al. [59], they reuse a pre-optimized IR rather than the final machine code. They defer VM-specific optimizations until the IR is compiled for a specific virtual machine. As

a consequence, one has to decide which optimizations can be performed ahead of time and which must be deferred to maintain performance. Thus, the heuristics they used do not decide which optimizations to perform, but when to perform them (i.e., the point of time within the compilation toolchain). In our case, the point of time when to select the execution mode is, however, fixed.

6.2.2 Static Branch Prediction

This technique offers advantages by estimating dynamic profiling data at compile time, relieving programmers from dynamic instrumentation and profiling. Wu and Larus [114] proposed several techniques and heuristics for static profiling. Calder et al. [14] introduced a machine learning-based approach to static branch prediction and could reduce the rate of branch prediction misses from 25% to 20%. They propose evidence-based static prediction (ESP), which uses existing programs to train neural networks and decision trees to predict the behavior of branches in new programs. Their conclusion is that neural networks perform better. The ESP method moves away from manually created, expert-defined heuristics and demonstrates a more data-driven way to develop compiler heuristics, achieving a lower miss rate for branch prediction than existing program-based heuristics.

While static branch prediction also deals with heuristics and execution behavior to optimize for certain goals such as performance, the heuristics of our research are about compilation units and less about traces and branches.

6.2.3 Optimizations Along the Trade-off Between Size and Performance

Debray et al. [22] perform optimizations to achieve more compact code by factoring out common parts. Their performance changes reach from 23% slowdown to 35% speedup, by a reduction of code size from 15% to 30%. However, our scenarios as described in Sections 3.1 and 5.1 differ, since the GraalVM compiler and/or clang, respectively, already perform these optimizations. Instead, we decide whether to compile a compilation unit or interpret it. Thus, our approach moves the decision to a different level and decides whether a compilation unit ends up being represented by native code or bytecode.

From the work we are aware of, the most closely related seem to be approaches to select which optimizations are to be performed to avoid increasing code size [30, 81]. Thus, such approaches look at which optimization levels or optimizations to use to achieve a certain code size. However, they still compile everything to native code for their use case and do not consider a hybrid execution system as we do here. Also, both contributions look at specific optimizations or whether to turn them on or off. Our approaches, however, require a general selection between two different execution modes. Thus, our selection is no single choice per optimization, but rather a choice between bundles of performed, indivisible optimizations at once. Applying code factorization as described by Debray et al. [22] to our approach of mixing execution modes sounds interesting on the first glimpse. In such a case, we expect the overall performance to depend on the overhead for switching the execution modes more often.

While machine learning in compilers is often used for specific optimizations, Cugurovic et al. [19] apply machine learning to predict the hotness of a method. They let hot code be highly optimized, while cold code is compiled to a less-optimized version. Using the fact that levels of optimization usually lead to larger executable binaries, they try to reduce the total code size while keeping performance as high as possible. Their concept of hybrid execution of code compiled with different optimization levels to reduce binary size inspires our approach in Chapter 3. Additionally, their implementation is based on GraalVM Native Image, an AOT compiler for Java bytecode. Their results based on machine learning outperform our static heuristics, as they achieve a reduction by 15–25% with a performance drop of 3% (5% with our static heuristics). However, their performance benefit comes with ML model training costs.

6.2.4 Optimization Selection

The approaches described in this subsection all deal with optimization selection, i.e., which optimization to use with regard to different goals. This is similar to our approaches of hybrid execution: While we do not combine different optimization levels in this sense, we combine different execution modes that come with different properties each, as described in Section 2.1.4.

Stephenson et al. [95] use an evolutionary algorithm to search the complex space of possible heuristics. Their approach can create specialized heuristics for specific applications, but can also generate effective, general-purpose heuristics that improve performance on unrelated test sets. Their work aims to relieve compiler designers from the difficult and tedious task of manually tuning heuristics, which is also the goal in Chapters 3 and 5. However, our work focuses on per-method / per-compilation-unit heuristics, while their approach proposes heuristics for optimizations like register allocation. Still, it can be worthwhile to apply and evaluate their approach of evolutionary algorithms to our code features.

Pinkers et al. [81] focus on the problem that standard compiler optimization levels, such as -O3, often produce suboptimal results because the best set of optimizations depends on the specific application. They propose an automatic, iterative method to select the best combination of compiler optimization flags. Their technique uses a statistical approach based on orthogonal arrays to analyze profile information and determine the individual impact of each optimization option. This allows for the creation of a custom set of optimizations for a given program, turning off specific optimizations that have been enabled for lower optimization levels. The authors show that their method outperforms the standard -O3 level in the GCC compiler for several SPEC benchmarks, with speed-ups between 3% and 26%.

Cavazos et al. [15] present a novel approach for selecting effective compiler optimizations by using hardware performance counters. Instead of relying on static code features or expensive iterative searches, the authors propose learning a model offline that maps performance counter data to good optimization settings. This trained model can then be used to quickly predict beneficial optimizations for any new program. The technique is shown to be two orders of magnitude faster than search-based approaches and achieves a 17% perfor-

mance improvement over the highest optimization level of a commercial compiler on the SPEC benchmark suite.

Hoste and Eeckhout [44] optimize along the trade-off between compilation time and execution time. They use a multi-objective evolutionary search to find combinations of standard compiler optimizations 00 to 03. They treat the selection of optimization flags as a search problem and apply an evolutionary algorithm as heuristic to automate and improve upon the hand-crafted heuristics for standard optimization levels.

As mentioned above, Cugurovic et al. [19] apply machine learning to predict the hotness of a method, which is later used for selecting a certain optimization level to reduce binary size. Interestingly, Cugurovic et al. [19] trained three separate models, one for each optimization level, and found that the most relevant method features varied across levels. This observation aligns with the finding by Pinkers et al. [81] that higher optimization levels do not merely accumulate the optimizations applied at lower levels, but may also discard some of them. Therefore, this leads to distinct optimization configurations at each level. This aligns with our results of Figure 3.24 in Section 3.6.5, where different heuristics perform best for different optimization levels.

Chapter 7

Future Work

In this chapter, we present possible improvements and future extensions to our current hybrid execution approaches. Much of the inspiration for our future work comes from the evaluation of our hybrid execution approaches as well as our related work. We will propose ways to extend our current capabilities and address several drawbacks.

7.1 Technical Implementation of Hybrid Execution

Regarding the technical implementation enabling hybrid execution, we consider the following aspects as sensible future work:

JIT Compiler The proposed hybrid system of Chapter 3 currently only supports AOT compilation and interpretation. In general, heuristics that select one of these two execution modes only provide an estimation for the better choice rather than the ground truth. As a consequence, there are performance-critical compilation units that are interpreted and thus cause a significant slowdown, as shown in Chapter 3. Thus, a JIT compiler could reduce the slowdown that such interpreted compilation units cause in the rare case they are relevant for peak performance.

The drawback of a JIT compiler, however, is its effect on warm-up performance. Nevertheless, we saw in Sections 3.5.4 and 3.6.4 that less than 5% of the interpreted compilation units in the DaCapo benchmarks would be candidates for JIT compilation, and even less, 2%, really cause slowdowns. Therefore, as the workload for a JIT compiler is rather low, we imagine the effects on warm-up performance to be low enough in practice. Thus, we suggest an evaluation of the actual effects on warm-up when a JIT compiler is used for the few performance-relevant compilation units that are interpreted.

While the approach in Chapters 4 and 5 already contains a JIT compiler, our heuristics try to minimize execution time. Another interesting metric for our heuristics, however, is to look at time compilation effort. Including an estimation for compilation time into heuristics can also help to reduce the time the application takes to reach peak performance. As an example, an estimation for code complexity within a compilation unit could be considered in this case.

Compilation Effort Currently, both systems compile the code AOT before deciding whether to run the compiled code or to go for interpretation. The compilation analysis and the compilation result provide useful information for our heuristics, such as the size of the compiled code or the estimated CPU cycles. However, performing analysis and compilation increases build time even when the heuristics later choose not to use the AOT-compiled code.

We therefore propose moving the decision earlier in the pipeline. Early rejection of units that are likely to be interpreted can reduce compilation effort and build time. For example, static initializers in Java (where we described in Section 3.2.4 why they should be interpreted) can already be identified at the byte-code level and skipped entirely. When savings of compilation time have higher priority than maximum peak performance, it can be preferable to use cheaper, less accurate heuristics that operate earlier to decide whether a compilation unit is worth compiling.

Dealing With Code That Can Only Run in Either Mode A limitation we encountered is that there are functions and compilation units that cannot run in both modes. As an example, a compilation unit of the Native Image runtime might be considered for interpretation, although it is required to be executed natively. Even more, such execution mode constraints cannot always be detected upfront: For native extensions, functions in native execution mode may access managed data which is neither statically known, nor can it be converted to native memory. When heuristics select a function for an impossible execution mode, run-time errors occur.

Thus, a future improvement is to reduce this risk of runtime errors by enabling to revert execution: Upon encountering an error, we could rewind the ongoing execution, forcing the problematic code to resume in a different execution mode. However, this includes a possibility to undo side effects, such as introducing and managing savepoints.

7.2 Heuristics

Besides possible technical implementation development, there is also future work in terms of developing more sophisticated heuristics:

Knowing Information Earlier During Compilation Process Our current approach only optimizes for the final binary size. However, it would be beneficial to optimize the compilation process in the future, too. While we can now select, which compilation units to include as native code or as bytecode in the final executable, we compile currently all compilation units during the build process, as described in Section 3.2.1. While this gives us the most complete information and allows us to make decisions based on knowing the final size of the compiled code, it does not reduce the overall compilation time. To also reduce the compilation time, future work could investigate the tradeoffs of moving the decision of which compilation units to compile to earlier stages of the process. This could allow us to also avoid the costly process of optimizing and creating the native code for compilation units where these results end up being discarded.

However, it is likely going to come at additional tradeoffs in terms of the quality of static predictions, and thus, may cause additional run-time slowdowns.

Static Heuristics for Call-Graph Estimation In Chapter 5, we used heuristics that rely on a call graph which we had to record in advance. Recording such information is expensive and requires at least one profiling run in advance. Thus, using static analysis information to estimate the call graph and its call frequencies at compile time would lead to multiple benefits: Not only would it become possible to avoid the profiling run, but also could decisions what to compile and what to interpret be made earlier, as mentioned above. In contrast to analyzing Java code as in Chapter 3, analyzing C code as in Chapter 5 can be done with a higher accuracy [49, 118].

More Sophisticated Heuristics: Machine Learning and Dynamic Profiling Stephenson et al. [95] showed that evolutionary algorithms can lead to impressive results when searching for combination of different optimization possibilities. Thus, it is worthwhile to investigate whether and how their approach can also lead to benefits within our search space of execution mode combinations for the set of compilation units.

Another future direction would be to combine our static heuristics with machine learning or dynamic profiling. We believe the static features could give strong initial predictions, and profiling or machine learning models would then only need to be used for compilation units with ambiguous static predictions or high performance impact. With such hybrid heuristics we could reach the performance of fully dynamic or machine-learning-driven systems, such as described by Cugurovic et al. [19], while requiring less training data or using fewer run-time profiling runs. Arnold et al. [2] previously showed how to efficiently combine such static and dynamic information for inlining.

Concerning our dynamic heuristics in Chapter 3, we saw in Section 3.6.5 that they can be outperformed by more advanced static heuristics. Therefore, we expect that a dynamic heuristic relying on more static information, i.e., the i_{per} heuristic or even more, leads to further improvements.

Heuristics For Warm-Up Performance Although an overall speed-up can be achieved, there are special functions where the automated hybrid execution causes a slowdown in warm-up performance. Therefore, further research is necessary to investigate the connection between single functions and the impact on warm-up performance when automated hybrid execution is enabled. Being able to predict the impact of hybrid execution on warm-up performance more precisely also enables better results overall.

Question of Mutual Applicability of Heuristics of Chapters 3 and 5 We analyzed two different scenarios for hybrid execution: Hybrid execution to reduce code size without impacting performance in Chapter 3, and hybrid execution of native extensions in Chapters 4 and 5 to improve performance. However, we did not mutually apply the used heuristics to the corresponding “other” use case.

While this is due to different contexts (e.g., high overhead costs for native extensions), implementation details of both systems might change. New technical circumstances, however, may limit the effectiveness of the currently used heuristics. Therefore, predicting heuristics based on implementation details would keep the achievements of the current systems, while enabling flexibility of maintenance.

Fitting Heuristics To Code The lxml evaluation in Section 5.4.1 shows that our heuristics in Chapter 5 are still specific to a certain kind of code. The lxml benchmark contains a lot of different computations and leads to different results for them. Thus, inspired by the work of Cugurovic et al. [19], it is interesting how well ML-based techniques can predict the optimal execution mode for a function.

Chapter 8

Conclusion

Modern high-performance runtimes are constrained by the fundamental trade-offs between peak performance, warm-up performance, and binary executable size. While AOT compilation provides fast warm-up, it often results in bloated binaries. JIT compilation optimizes peak performance but introduces unpredictable warm-up costs. Existing strategies for mitigating these issues, such as the use of native extensions, require a mechanism to change execution mode during an ongoing execution. Furthermore, existing mixed-mode systems requires the developer to manually select the execution mode for every part of the application. This does not only lead to suboptimal results, but also requires knowledge of execution constraints, as usually not every code can be executed in any of the available execution modes.

In this thesis, we addressed these limitations by implementing and evaluating automated hybrid execution models on top of GraalVM, a high-performance, production-grade and multi-language runtime that is also widely used in industry. To reduce the size of binary executables and solve goal 1 mentioned in Section 1.2, we implemented a mechanism that replaces selected AOT-compiled code by its more compact Java bytecode. The bytecode is then executed by a lightweight interpreter. To optimize mainly warm-up, but also peak performance of polyglot applications, we introduced a hybrid environment for native extensions that enables switching between managed and native execution, tackling goal 2.

The selection of execution modes on both scenarios was automated through a suite of static and dynamic heuristics, which refers to our goal 3. While successful static heuristics are based on features such as loop nesting, number of parameters, estimated cycles, dynamic heuristics perform better by taking the call frequency into account. In both scenarios, we could show that initialization code behaves different compared to average code segments concerning our objectives, such as performance and machine code size. With these heuristics, we show how to automate such a decision process, that aims to find a suitable choice along the corresponding trade-offs between warm-up performance, peak performance and application size.

Our evaluations of these heuristics shows that our hybrid approaches are both feasible and effective in state-of-the-art runtimes. For binary size reduction, we demonstrated that it is possible to reduce the size of a binary executable by over 35% without significant performance changes. Using only statically available information, our heuristics can lead to an overall size reduction by 20% with an average slowdown of only 4% on DaCapo and Renaissance benchmarks. Thus, our heuristics can be used to tune an application along the trade-off be-

tween performance and size. Regarding native extensions, our automated hybrid execution improved especially warm-up performance by a factor of $1.7\times$ to $2.5\times$ over fully managed baselines for lxml and ujson benchmarks. For peak performance, the result differs and depends on the code, as we could achieve speed-ups of $3.5\times$, but also had slowdowns of 15% compared to a fully-managed baseline. Our measurements on ujson show that automated hybrid execution can also outperform manual execution mode selection. Although the overall effectiveness of these heuristics can be influenced by specific code semantics and implementation details, our results indicate that an automated mixing of execution modes is a useful strategy for balancing the trade-offs of modern runtimes.

List of Figures

1.1	Three execution modes (AOT compilation, JIT compilation and interpretation) and their advantages.	2
2.1	Basic structure of an AOT compiler: Source code is transformed into binary machine code. A common, but optional step is to use intermediate representations in between to enable optimizations.	10
2.2	Basic procedure of a JIT compiler combined with an interpreter: The interpreter starts to execute the program. Profiling can detect often-executed code or paths within the code, and can compile them to optimized machine code. In case of a wrong assumption, code can be deoptimized, and execution reverts back to interpretation again.	11
2.3	Three execution modes (AOT compilation, JIT compilation and interpretation) and their advantages, as in Figure 1.1.	12
2.4	Static and Dynamic Linking in comparison.	17
2.5	Dynamic function resolution using PLT and GOT.	17
2.6	GraalVM ecosystem: Java bytecode can be used directly, whereas other languages require a runtime based on the Truffle Language Implementation Framework. The GraalVM compiler can serve as a JIT compiler for Java bytecode, while GraalVM Native Image compiles Java bytecode to machine code ahead of time.	20
2.7	The Sulong Toolchain offers compilation of C/C++ code to machine code and includes the corresponding LLVM bitcode as well.	24
2.8	GraalVM Native Image compilation process from code perspective: Method A gets compiled regularly. Method C gets inlined into B. Method D could be proven to be dead code, so it is not included into the final executable.	26
2.9	Native Image build process [117]: Java bytecode, including runtime code, is analyzed and optimized and then compiled to an executable.	26
2.10	Sample data values and their box plot with box, whiskers and one outlier.	28
3.1	Compilation process of GraalVM Native Image, with and without our approach.	32
3.2	Comparing code sizes: bytecode vs. machine code. On average, machine code of a compilation unit is larger than its bytecode.	33
3.3	Distribution of the ratio $\frac{\text{interpreter execution time}}{\text{AOT execution time}}$ for compilation units of the DaCapo <i>fop</i> benchmark. It includes all compilation units activated at least once. It shows that for more than 90%, the benefit of AOT compilation is larger than a factor 400.	35
3.4	Estimated computational density of cycles per byte compared to the execution time ratio. Higher density, i.e., more estimated CPU cycles per size benefits more from AOT compilation.	36

3.5	The number of parameters of a compilation unit compared to the execution time ratio. The more parameters a compilation unit has, the more compilation pays off.	37
3.6	Activation Times of Compilation Units: There are some compilation units (marked orange) whose first activation takes much longer than subsequent ones. The y value is computed as in Equation 3.1.	38
3.7	Scatter plot showing the early return ratio compare to the execution time ratio. The data suggests that the earlier a return can be reached, the more compilation pays off.	39
3.8	Scatter plot showing the number of null checks compare to the execution time ration. More null seem to indicate a higher benefit of compilation for compilation unit.	40
3.9	Usages of static initializers: In the DaCapo benchmarks, only 12% to 20% of the static initializers are actually called.	41
3.10	Sizes of Static Initializers in DaCapo Benchmarks: On average, static initializers are larger than an average method.	42
3.11	Native Image Build for Hybrid Execution.	47
3.12	Indirection step via GOT/PLT allows dynamic resolution of method calls.	48
3.13	Scatter plot of compilation units of an application where only a few are performance-relevant. With a higher x value, more units are interpreted. Critical units can be detected when the run time increases, as marked in red.	50
3.14	Comparison: Time spent in the interpreter vs. total execution time, each dot is a compilation unit. Starting from 10^8 ns in the interpreter, interpretation time also affects total execution time of the benchmark.	51
3.15	Necessary size savings by code removal to achieve smaller executables when interpreter and a JIT compiler are included.	52
3.16	Single-feature results, grouped by size reduction and heuristics. The heuristic based on number of parameters (pars) achieves the size reduction with the lowest slowdown, and thus, is the best individual heuristic.	55
3.17	Combined results, grouped by size reduction and heuristic. Overall, the iper combination of heuristics achieves the size reduction with the lowest overhead.	56
3.18	Trade-off between size and performance for our iper heuristics. At a size reduction of 35% we see a worst case slowdown of less than 20% with the median slowdown staying under 7%.	57
3.19	Trade-off between size and performance per heuristic. Showing the median slowdown. iper is the overall best candidate, while the random heuristics are causing the highest slowdowns overall.	57
3.20	Detailed benchmark results for a 20% size reduction using the iper heuristic. The highest slowdown is caused on fop, while most Renaissance benchmarks see only minor slowdowns.	59
3.21	Run with sorted order and dynamic heuristic – DaCapo fop.	60
3.22	DaCapo Benchmarks with dynamic heuristic: Execution time and ratio of JIT-compiled code.	61
3.23	Achieved code savings of different benchmarks. A point above the line denotes a total saving of size for the corresponding benchmark.	64

3.24	Comparing static and dynamic heuristics with a designated size reduction. Dynamic heuristics detect uncalled compilation units and thus perform better in most cases. They do not even cause a slowdown as long as uncalled compilation units are removed. . . .	65
4.1	Applications with Native Extensions can be executed in different ways. Nodes denote functions, while arrows denote calls or accesses.	70
4.2	The compilation result of the Sulong toolchain includes LLVM bitcode for managed execution as well as native code within the same file. Thus, the compilation result includes two different versions of the source code at the same time.	72
4.3	Memory object handling in different scenarios.	73
4.4	Lazy conversion to native memory with a C array as an example: Managed array (Fig. 4.4a), array passed to native execution (Fig. 4.4b), and array elements being converted due to access of individual elements (Fig. 4.4c).	75
4.5	Symbol access within the same file. However, such an access within the same file, but from managed to native execution, looks like a foreign access from the perspective of dlopen/dlsym.	77
4.6	Calculation of the run-time address of a symbol in native memory.	78
4.7	If foo should be executed native, then its call node in the LLVM Truffle AST has to be modified.	79
4.8	Hybrid execution models for function a calling b, b calling c, and c accessing managed data. Dotted lines and gray nodes indicate that the call/method cannot be used.	80
4.9	A change in the procedure linkage table (PLT) can make native callers call managed callees.	81
4.10	Functions may get duplicated — Call structure (Figure 4.10a) and a possible execution scenario (Figure 4.10b).	82
5.1	Execution time of a function called from Sulong without branches and loops, based on its code size of LLVM bitcode.	87
5.2	The maximum loop nesting level of Listing 5.1 is calculated.	88
5.3	Function call overhead in different execution mode scenarios.	90
5.4	Function f can be executed in managed mode as well as in native mode. Since overhead costs for upcalls and downcalls of f also influence execution time, they are taken into account.	91
5.5	Results of LXML Objectify Benchmark: For data setup and warm-up performance, native execution performs better. For annotation tasks, our hybrid mode pays off. For path search within the tree, managed execution performs best.	94
5.6	Results of the UJSON Benchmark: Our automated heuristics for hybrid execution achieve an overall speedup.	96

List of Tables

- 2.1 Overview of existing compiler heuristics and corresponding im-
portant publications on which we build upon. 14
- 3.1 Basic principle of hybrid execution. 33
- 3.2 Compilation unit features and their heuristic score functions. . . . 45
- 3.3 Sample compilation units and their scores for ipe heuristics. 46
- 3.4 Native Image Executables of DaCapo Benchmarks. 58
- 3.5 Possible sweet spot results of DaCapo benchmarks, sorted by total
image size changes. 62

Listings

2.1	<code>pointParser.js</code> – JavaScript code that parses a string and returns a JS object.	22
2.2	<code>pointLib.c</code> – C code that modifies fields of a point object. The code is then compiled to LLVM IR/machine code (<code>pointLib.so</code>). . .	22
2.3	<code>main.py</code> – Example for Truffle interoperability: Python code that calls JavaScript code (Listing 2.1) to parse and C code (Listing 2.2) to modify the parsed data.	23
3.1	Lazy Initialization Pattern in Java. A presumed candidate pattern for detection.	38
4.1	C code containing two functions, where one is called from the other. The function <code>fetch_point</code> accesses a global variable.	78
4.2	Problem: callee is still executed in native mode.	80
4.3	Simple, but not feasible workaround to call managed functions from native code is to store every managed function in an explicit pointer variable. For that, <code>init</code> has to be executed in managed mode at the start of the program.	81
5.1	Sample code containing loops.	88

List of Acronyms

AOT	ahead of time
API	application programming interface
AST	abstract syntax tree
BFS	breadth-first search
DFS	depth-first search
ELF	executable and linkable format
ESP	evidence-based static prediction
FDO	feedback-directed optimization
FFI	foreign function interface
FFM	foreign function and memory
FOP	formatting object processor
GC	garbage collector
GCC	GNU compiler collection
GOT	global offset table
IDE	integrated development environment
IR	intermediate representation
JDK	Java development kit
JIT	just in time
JNI	Java native interface
JS	JavaScript
JSON	JavaScript object notation
JVM	Java virtual machine
JVMCI	Java virtual machine compiler interface
LLVM	a cross-platform compilation framework (formerly known as Low-Level Virtual Machine)
ML	machine learning
NFI	native function interface
OSR	on-stack replacement
PGO	profile-guided optimization
PLT	procedure linkage table
SSA	static single assignment
VM	virtual machine
XML	extensible markup language

Bibliography

- [1] Varun Agrawal, Abhiroop Dabral, Tapti Palit, Yongming Shen, and Michael Ferdman. 2015. Architectural Support for Dynamic Linking. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. Association for Computing Machinery, Istanbul, Turkey, pp. 691–702. ISBN: 9781450328357. DOI: 10.1145/2694344.2694392.
- [2] Matthew Arnold, Stephen Fink, Vivek Sarkar, and Peter F. Sweeney. 2000. A comparative study of static and profile-based heuristics for inlining. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (DYNAMO '00)*. Association for Computing Machinery, New York, NY, USA, pp. 52–64. ISBN: 1581132417. DOI: 10.1145/351397.351416.
- [3] John Aycock. 2003. A Brief History of Just-in-Time. *ACM Comput. Surv.*, 35, 2, (June 2003), 97–113. ISSN: 0360-0300. DOI: 10.1145/857076.857077.
- [4] Michael Backes, Sven Bugiel, Oliver Schranz, Philipp Von Styp-Rekowsky, and Sebastian Weisgerber. 2017. ARTist: The Android Runtime Instrumentation and Security Toolkit. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 481–495. DOI: 10.1109/EuroSP.2017.43.
- [5] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. 1957. The FORTRAN Automatic Coding System. In *Papers Presented at the February 26–28, 1957, Western Joint Computer Conference: Techniques for Reliability (IRE-AIEE-ACM '57 (Western))*. ACM, Los Angeles, California, pp. 188–198. ISBN: 9781450378611. DOI: 10.1145/1455567.1455599.
- [6] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. 1994. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26, 4, (December 1994), 345–420. ISSN: 0360-0300. DOI: 10.1145/197405.197406.
- [7] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual machine warmup blows hot and cold. *Proc. ACM Program. Lang.*, 1, OOPSLA, Article 52, (October 2017), 27 pages. DOI: 10.1145/3133876.
- [8] Aurèle Barrière, Sandrine Blazy, Olivier Flückiger, David Pichardie, and Jan Vitek. 2021. Formally verified speculation and deoptimization in a JIT compiler. *Proc. ACM Program. Lang.*, 5, POPL, Article 46, (January 2021), 26 pages. DOI: 10.1145/3434327.

- [9] Matteo Basso, Aleksandar Prokopec, Andrea Rosà, and Walter Binder. 2025. Improving Native-Image Startup Performance. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO '25)*. Association for Computing Machinery, Las Vegas, NV, USA, pp. 689–703. ISBN: 9798400712753. DOI: 10.1145/3696443.708927.
- [10] Monica Beckwith. 2024. *Jvm performance engineering: inside openjdk and the hotspot java virtual machine*. Addison-Wesley Professional. ISBN: 9780134659879.
- [11] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. Association for Computing Machinery, Portland, Oregon, USA, pp. 169–190. ISBN: 1595933484. DOI: 10.1145/1167473.1167488.
- [12] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the Meta-Level: PyPy's Tracing JIT Compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS '09)*. Association for Computing Machinery, Genova, Italy, pp. 18–25. ISBN: 9781605585413. DOI: 10.1145/1565824.1565827.
- [13] Marc M. Brandis and Hanspeter Mössenböck. 1994. Single-pass generation of static single-assignment form for structured languages. *ACM Trans. Program. Lang. Syst.*, 16, 6, (November 1994), 1684–1698. ISSN: 0164-0925. DOI: 10.1145/197320.197331.
- [14] Brad Calder, Dirk Grunwald, Michael Jones, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn. 1997. Evidence-based static branch prediction using machine learning. *ACM Trans. Program. Lang. Syst.*, 19, 1, (January 1997), 188–222. ISSN: 0164-0925. DOI: 10.1145/239912.239923.
- [15] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F.P. O'Boyle, and Olivier Temam. 2007. Rapidly Selecting Good Compiler Optimizations using Performance Counters. In *International Symposium on Code Generation and Optimization (CGO'07)*, pp. 185–197. DOI: 10.1109/CGO.2007.32.
- [16] 2025. Clang: a C language family frontend for LLVM. (December 2025). <https://clang.llvm.org/>.
- [17] 2025. clang: C++ compiler. (December 2025). <https://www.clang.org/>.
- [18] Cliff Click and Michael Paleczny. 1995. A simple graph-based intermediate representation. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations (IR '95)*. Association for Computing Machinery, San Francisco, California, USA, pp. 35–49. ISBN: 0897917545. DOI: 10.1145/202529.202534.

- [19] Milan Cugurovic, Aleksandar Prokopec, Boris Spasojevic, Vojin Jovanovic, and Milena Vujošević Janičić. 2026. GraalMHC: ML-Based Method-Hotness Classification for Binary-Size Reduction in Optimizing Compilers. In *Proceedings of the 35th ACM SIGPLAN International Conference on Compiler Construction (CC '26)*. Association for Computing Machinery, Sydney, NSW, Australia, pp. 1–13. ISBN: 9798400722745. DOI: 10.1145/3771775.3786276.
- [20] cython. 2025. Cython: C-Extensions for Python. (December 2025). <https://cython.org/>.
- [21] 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, (October 1991), 451–490. ISSN: 0164-0925. DOI: 10.1145/115372.115320.
- [22] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. 2000. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst.*, 22, 2, (March 2000), 378–415. ISSN: 0164-0925. DOI: 10.1145/349214.349233.
- [23] David Detlefs and Ole Agesen. 1999. Inlining of Virtual Methods. In *ECOOP' 99 — Object-Oriented Programming*. Rachid Guerraoui, (Ed.) Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 258–277. ISBN: 978-3-540-48743-2.
- [24] Gilles Duboscq, Lukas Stadler, Thomas Wuerthinger, 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*. (February 2013).
- [25] Gilles Duboscq, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Speculation without regret: reducing deoptimization meta-data in the Graal compiler. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '14)*. Association for Computing Machinery, Cracow, Poland, pp. 187–193. ISBN: 9781450329262. DOI: 10.1145/2647508.2647521.
- [26] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages (VMIL '13)*. Association for Computing Machinery, Indianapolis, Indiana, USA, pp. 1–10. ISBN: 9781450326018. DOI: 10.1145/2542142.2542143.
- [27] Instagram Engineering. 2016. Continuous Deployment at Instagram — [instagram-engineering.com](https://instagram-engineering.com/continuous-deployment-at-instagram-1e18548f01d1). <https://instagram-engineering.com/continuous-deployment-at-instagram-1e18548f01d1>. [Accessed 01-08-2025]. (2016).
- [28] Shopify Engineering. 2016. Automatic Deployment at Shopify — [shopify.engineering](https://shopify.engineering/automatic-deployment-at-shopify). <https://shopify.engineering/automatic-deployment-at-shopify>. [Accessed 01-08-2025]. (2016).

- [29] Olivier Flückiger, Gabriel Scherer, Ming-Ho Yee, Aviral Goel, Amal Ahmed, and Jan Vitek. 2017. Correctness of speculative optimizations with dynamic deoptimization. *Proc. ACM Program. Lang.*, 2, POPL, Article 49, (December 2017), 28 pages. DOI: 10.1145/3158137.
- [30] Juliano Henrique Foleiss, Anderson Faustino da Silva, and Linnyer Beatrys Ruiz. 2011. The Effect of Combining Compiler Optimizations on Code Size. In *2011 30th International Conference of the Chilean Computer Science Society*, pp. 187–194. DOI: 10.1109/SCCC.2011.25.
- [31] Elisa Fröhlich, Angélica Moreira, and Fernando Magno Quintão Pereira. 2026. Automatic Propagation of Profile Information through the Optimization Pipeline. *Proc. ACM Program. Lang.*, 10, OOPSLA1, Article 139, (April 2026), 26 pages. DOI: 10.1145/3798247.
- [32] Yoshihiko Futamura. 1983. Partial computation of programs. In *RIMS Symposia on Software Science and Engineering*. Eiichi Goto, Koichi Furukawa, Reiji Nakajima, Ikuo Nakata, and Akinori Yonezawa, (Eds.) Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 1–35. ISBN: 978-3-540-39442-6.
- [33] GNU. [n. d.] GCC, the GNU Compiler Collection. <https://gcc.gnu.org/>. [Accessed 30-04-2026]. ().
- [34] Anjana Gosain and Ganga Sharma. 2015. A Survey of Dynamic Program Analysis Techniques and Tools. In *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014*. Suresh Chandra Satapathy, Bhabendra Narayan Biswal, Siba K. Udgata, and J.K. Mandal, (Eds.) Springer International Publishing, Cham, pp. 113–122. ISBN: 978-3-319-11933-5.
- [35] graalvm.org – javadoc. 2025. Class CPUSampler. (December 2025). <https://www.graalvm.org/tools/javadoc/com/oracle/truffle/tools/profiler/CPUSampler.html>.
- [36] Matthias Grimmer, Manuel Rigger, Lukas Stadler, Roland Schatz, and Hanspeter Mössenböck. 2013. An efficient native function interface for Java. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '13)*. Association for Computing Machinery, Stuttgart, Germany, pp. 35–44. ISBN: 9781450321112. DOI: 10.1145/2500828.2500832.
- [37] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. 2015. High-performance cross-language interoperability in a multi-language runtime. In *Proceedings of the 11th Symposium on Dynamic Languages (DLS 2015)*. Association for Computing Machinery, Pittsburgh, PA, USA, pp. 78–90. ISBN: 9781450336901. DOI: 10.1145/2816707.2816714.
- [38] Matthias Grimmer, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. 2015. Dynamically composing languages in a modular way: supporting C extensions for dynamic languages. In *Proceedings of the 14th International Conference on Modularity (MODULARITY 2015)*. Association for Computing Machinery, Fort Collins, CO, USA, pp. 1–13. ISBN: 9781450332491. DOI: 10.1145/2724525.2728790.

- [39] Thomas Grurl, Christoph Pichler, Jürgen Fuß, and Robert Wille. 2023. Automatic Implementation and Evaluation of Error-Correcting Codes for Quantum Computing: An Open-Source Framework for Quantum Error Correction. In *2023 36th International Conference on VLSI Design and 2023 22nd International Conference on Embedded Systems (VLSID)*, pp. 301–306. DOI: 10.1109/VLSID57277.2023.00068.
- [40] Wenlei He, Julián Mestre, Sergey Pupyrev, Lei Wang, and Hongtao Yu. 2022. Profile inference revisited. *Proc. ACM Program. Lang.*, 6, POPL, Article 52, (January 2022), 24 pages. DOI: 10.1145/3498714.
- [41] Michael Hicks, Stephanie Weirich, and Karl Crary. 2001. Safe and Flexible Dynamic Linking of Native Code. In *Types in Compilation*. Robert Harper, (Ed.) Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 147–176. ISBN: 978-3-540-45332-1.
- [42] 2023. High-performance Modern Python – run your applications with GraalPy. (May 2023). <https://www.graalvm.org/python/>.
- [43] Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI '92)*. Association for Computing Machinery, San Francisco, California, USA, pp. 32–43. ISBN: 0897914759. DOI: 10.1145/143095.143114.
- [44] Kenneth Hoste and Lieven Eeckhout. 2008. Cole: compiler optimization level exploration. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '08)*. Association for Computing Machinery, Boston, MA, USA, pp. 165–174. ISBN: 9781595939784. DOI: 10.1145/1356058.1356080.
- [45] 2023. HPy: A better API for Python. (June 2023). <https://docs.hpyproject.org/en/latest/>.
- [46] Apple Inc. [n. d.] Swift Programming Language. <https://www.swift.org>. [Accessed 30-04-2026]. ().
- [47] Oracle Inc. 2025. HotSpot Virtual Machine Garbage Collection Tuning Guide.
- [48] Michael R. Jantz and Prasad A. Kulkarni. 2013. Exploring single and multilevel JIT compilation policy for modern machines 1. *ACM Trans. Archit. Code Optim.*, 10, 4, Article 22, (December 2013), 29 pages. ISSN: 1544-3566. DOI: 10.1145/2541228.2541229.
- [49] Kinloch and Munro. 1994. Understanding C programs using the Combined C Graph representation. In *Proceedings 1994 International Conference on Software Maintenance*, pp. 172–180. DOI: 10.1109/ICSM.1994.336777.
- [50] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for life-long program analysis and transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. Pp. 75–86. DOI: 10.1109/CGO.2004.1281665.
- [51] Hugh Leather and Chris Cummins. 2020. Machine Learning in Compilers: Past, Present and Future. In *2020 Forum for Specification and Design Languages (FDL)*, pp. 1–8. DOI: 10.1109/FDL50818.2020.9232934.

- [52] 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 2018)*. Association for Computing Machinery, Boston, MA, USA, pp. 26–35. ISBN: 9781450360715. DOI: 10.1145/3281287.3281290.
- [53] libffi. 2025. libffi/libffi – a portable foreign function interface library. (December 2025). <https://github.com/libffi/libffi>.
- [54] Bingxin Liu, Yinghui Huang, Jianhua Gao, Jianjun Shi, Yongpeng Liu, Yipin Sun, and Weixing Ji. 2025. From Profiling to Optimization: Unveiling the Profile Guided Optimization. (2025). arXiv: 2507.16649 [cs.PF]. <https://arxiv.org/abs/2507.16649>.
- [55] P. Louridas. 2006. Static code analysis. *IEEE Software*, 23, 4, 58–61. DOI: 10.1109/MS.2006.114.
- [56] lxml. 2024. lxml/lxml – The lxml XML toolkit for Python. (2024). <https://github.com/lxml/lxml>.
- [57] man7.org. 2026. dlopen(3) — Linux manual page. (June 2026). <https://www.man7.org/linux/man-pages/man3/dlopen.3.html>.
- [58] man7.org. 2026. dlsym(3) — Linux manual page. (June 2026). <https://www.man7.org/linux/man-pages/man3/dlsym.3.html>.
- [59] Meetesh Kalpesh Mehta, Sebastián Krynski, Hugo Musso Gualandi, Manas Thakur, and Jan Vitek. 2023. Reusing Just-in-Time Compiled Code. *Proc. ACM Program. Lang.*, 7, OOPSLA2, Article 263, (October 2023), 22 pages. DOI: 10.1145/3622839.
- [60] Microsoft. 2025. .NET: Native AOT deployment. (December 2025). <https://learn.microsoft.com/en-us/dotnet/core/deploying/native-aot>.
- [61] Lazar Milikic, Milan Cugurovic, and Vojin Jovanovic. 2025. GraalNN: Context-Sensitive Static Profiling with Graph Neural Networks. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO '25)*. Association for Computing Machinery, Las Vegas, NV, USA, pp. 123–136. ISBN: 9798400712753. DOI: 10.1145/3696443.3708958.
- [62] Raphael Mosaner, David Leopoldseder, Wolfgang Kisling, Lukas Stadler, and Hanspeter Mössenböck. 2022. Machine-Learning-Based Self-Optimizing Compiler Heuristics. In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes (MPLR '22)*. Association for Computing Machinery, Brussels, Belgium, pp. 98–111. ISBN: 9781450396967. DOI: 10.1145/3546918.3546921.
- [63] Hanspeter Mössenböck. 2024. *Compiler Construction: Fundamentals and Applications*. Springer Nature Switzerland. ISBN: 9783031848131. DOI: 10.1007/978-3-031-84813-1.
- [64] Hanspeter Mössenböck. 2024. Dynamic Compiler Optimizations in the Graal VM. In *2024 IEEE 17th International Scientific Conference on Informatics (Informatics)*, pp. 1–1. DOI: 10.1109/Informatics62280.2024.10900774.

- [65] Hanspeter Mössenböck and Matthias Grimmer. 2015. Truffle—A Self-optimizing Language Implementation Framework. *5th Int. Conf. on Mathematics and Informatics*.
- [66] Peter W. O’Hearn and Robert D. Tennent, (Eds.) 1997. *Revised Report on the Algorithmic Language Algol 60. Algol-like Languages*. Birkhäuser Boston, Boston, MA, pp. 19–49. ISBN: 978-1-4612-4118-8. DOI: 10.1007/978-1-4612-4118-8_2. https://doi.org/10.1007/978-1-4612-4118-8_2.
- [67] NumPy. 2025. NumPy. (December 2025). <https://numpy.org/>.
- [68] OpenJDK. 2024. JEP 454: Foreign Function & Memory API. <https://openjdk.org/jeps/454>. [Accessed 30-06-2026]. (2024).
- [69] Oracle. 2022. Java Debug Wire Protocol (JDWP) with Native Image — docs.oracle.com. <https://github.com/oracle/graal/blob/master/docs/reference-manual/native-image/JDWP.md>. [Accessed 04-03-2025]. (2022).
- [70] Oracle. 2026. Optimize a Native Executable with Profile-Guided Optimization. <https://www.graalvm.org/latest/reference-manual/native-image/guides/optimize-native-executable-with-pgo/>. [Accessed 12-05-2026]. (2026).
- [71] Oracle. 2023. Profiling Truffle Interpreters. (June 2023). <https://docs.oracle.com/en/graalvm/jdk/22/docs/graalvm-as-a-platform/language-implementation-framework/Profiling/>.
- [72] Oracle. 2023. Substrate VM Project — docs.oracle.com. <https://docs.oracle.com/en/graalvm/enterprise/20/docs/reference-manual/native-image/SubstrateVM>. [Accessed 19-02-2025]. (2023).
- [73] Guilherme Ottoni and Bin Liu. 2021. HHVM Jump-Start: Boosting Both Warmup and Steady-State Performance at Scale. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 340–350. DOI: 10.1109/CGO51591.2021.9370314.
- [74] Andrej Pečimúth, David Leopoldseder, and Petr Tůma. 2024. An Analysis of Compiled Code Reusability in Dynamic Compilation. In *Proceedings of the 16th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL ’24)*. Association for Computing Machinery, Pasadena, CA, USA, pp. 43–53. ISBN: 9798400712135. DOI: 10.1145/3689490.3690406.
- [75] Christoph Pichler. 2022. *Running Swift-compiled LLVM IR code on Sulong*. eng. Master’s thesis. Johannes Kepler University Linz. <https://resolver.obvsg.at/urn:nbn:at:at-ubl:1-49289>.
- [76] Christoph Pichler, Paley Li, Roland Schatz, and Hanspeter Mössenböck. 2023. Hybrid Execution: Combining Ahead-of-Time and Just-in-Time Compilation. In *Proceedings of the 15th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL 2023)*. Association for Computing Machinery, Cascais, Portugal, pp. 39–49. ISBN: 9798400704017. DOI: 10.1145/3623507.3623554.

- [77] Christoph Pichler, Paley Li, Roland Schatz, and Hanspeter Mössenböck. 2023. Interoperability between Execution Modes on GraalVM. In *ACM Student Research Competition (SRC) at IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. <https://ssw.jku.at/Research/Papers/Pichler23/Pichler23SRC.pdf>.
- [78] Christoph Pichler, Paley Li, Roland Schatz, and Hanspeter Mössenböck. 2024. On Automating Hybrid Execution of Ahead-of-Time and Just-in-Time Compiled Code. In *Proceedings of the 16th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '24)*. Association for Computing Machinery, Pasadena, CA, USA, pp. 1–11. ISBN: 9798400712135. DOI: 10.1145/3689490.3690398.
- [79] Christoph Pichler, Bernhard Urban-Forster, Paley Li, Roland Schatz, and Stefan Marr. 2026. To Compile or Not To Compile: Evaluating Static Heuristics to Reduce Binary Size of Hybrid Execution Systems. In *Proceedings of the 23rd ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '26)*. Association for Computing Machinery, Brussels, Belgium. DOI: 10.1145/3828170.3828171.
- [80] Christoph Pichler, Bernhard Urban-Forster, Paley Li, Roland Schatz, and Hanspeter Mössenböck. 2025. Fast and Compact: Reducing Size of AOT-Compiled Java Code without Sacrificing Performance. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '25)*. Association for Computing Machinery, Singapore, Singapore, pp. 12–22. ISBN: 9798400721496. DOI: 10.1145/3759426.3760976.
- [81] R.P.J. Pinkers, P.M.W. Knijnenburg, M. Haneda, and H.A.G. Wijshoff. 2004. Statistical selection of compiler options. In *The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004. (MASCOTS 2004). Proceedings*. Pp. 494–501. DOI: 10.1109/MASCOT.2004.1348305.
- [82] 2026. Project Crema: Open World for Native Image. (April 2026). <https://github.com/oracle/graal/issues/11327>.
- [83] 2026. Project Leyden: Improve Java startup. (July 2026). <https://developer.ibm.com/articles/java-project-leyden/>.
- [84] Aleksandar Prokopec, Gilles Duboscq, David Leopoldseder, and Thomas Wirthinger. 2019. An Optimization-Driven Incremental Inline Substitution Algorithm for Just-in-Time Compilers. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 164–179. DOI: 10.1109/CGO.2019.8661171.
- [85] 2023. PyPy – A fast, compliant alternative implementation of Python. (June 2023). <https://www.pypy.org/>.
- [86] python. 2025. python/cpython – The Python programming language. (December 2025). <https://github.com/python/cpython>.
- [87] python. 2025. The Python programming language. (December 2025). <https://www.python.org/>.
- [88] python.org. 2024. PEP 744 – JIT Compilation. (April 2024). <https://peps.python.org/pep-0744/>.

- [89] 2025. Python/C API Reference Manual. (December 2025). <https://docs.python.org/3/c-api/index.html>.
- [90] Stephan Richter. 2024. lmxl – Benchmarks and Speed. (2024). <https://lxml.de/performance.html>.
- [91] Manuel Rigger, Matthias Grimmer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. 2016. Bringing low-level languages to the JVM: efficient execution of LLVM IR on Truffle. In *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages (VMIL 2016)*. Association for Computing Machinery, Amsterdam, Netherlands, pp. 6–15. ISBN: 9781450346450. DOI: 10.1145/2998415.2998416.
- [92] Manuel Rigger, Roland Schatz, René Mayrhofer, Matthias Grimmer, and Hanspeter Mössenböck. 2018. Sulong, and Thanks for All the Bugs: Finding Errors in C Programs by Abstracting from the Native Execution Model. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Williamsburg, VA, USA, pp. 377–391. DOI: 10.1145/3173162.3173174.
- [93] Manuel Serrano. 2021. Of JavaScript AOT compilation performance. *Proc. ACM Program. Lang.*, 5, ICFP, Article 70, (August 2021), 30 pages. DOI: 10.1145/3473575.
- [94] Marko Spasic, Miloje Joksimović, Peter Hofer, and Milena Vujošević Janičić. 2026. Code Layout Optimization Guided by Temporal Profiling Information. In *Companion Proceedings of the 10th International Conference on the Art, Science, and Engineering of Programming (<Programming> Companion '26)*. Association for Computing Machinery, pp. 9–13. ISBN: 9798400725555. DOI: 10.1145/3801119.3801124.
- [95] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. 2003. Meta optimization: improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03)*. Association for Computing Machinery, San Diego, California, USA, pp. 77–90. ISBN: 1581136625. DOI: 10.1145/781131.781141.
- [96] Tensorflow. 2025. Tensorflow. (December 2025). <https://www.tensorflow.org/>.
- [97] 2025. The Computer Language Benchmarks Game. (December 2025). <https://madnight.github.io/benchmarksgame/>.
- [98] 2025. The LLVM Compiler Infrastructure. (December 2025). <https://llvm.org/>.
- [99] 2026. The Python Performance Benchmark Suite. (2026). <https://pyperformance.readthedocs.io/>.
- [100] L. Torvalds. 2026. Linux Kernel. <https://github.com/torvalds/linux/tree/master/tools/perf>. [Accessed 11-03-2026]. (2026).
- [101] 2025. Truffle Native Function Interface. (December 2025). <https://www.graalvm.org/latest/graalvm-as-a-platform/language-implementation-framework/NFI/>.
- [102] UJSON. 2024. UltraJSON – Ultra fast JSON encoder and decoder for Python. (2024). <https://pypi.org/project/ujson/>.

- [103] UJSON. 2024. ultrajson/ultrajson – Ultra fast JSON decoder and encoder written in C with Python bindings. (2024). <https://github.com/ultrajson/ultrajson>.
- [104] Lionel Sujay Vailshery. 2025. Most used languages among software developers globally 2022. (December 2025). <https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/>.
- [105] Vincent Van Der Leun. 2017. *Introduction to JVM Languages*. Packt Publishing Ltd. ISBN: 9781787127944.
- [106] A. Van Hoff. 1997. The case for Java as a programming language. *IEEE Internet Computing*, 1, 1, 51–56. DOI: 10.1109/4236.585172.
- [107] Guido Van Rossum and Fred L. Drake. 2009. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA. ISBN: 1441412697.
- [108] Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2019. Detection of suspicious time windows in memory monitoring. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR 2019)*. Association for Computing Machinery, Athens, Greece, pp. 95–104. ISBN: 9781450369770. DOI: 10.1145/3357390.3361025.
- [109] Kevin Williams, Jason McCandless, and David Gregg. 2010. Dynamic interpretation for dynamic scripting languages. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '10)*. Association for Computing Machinery, Toronto, Ontario, Canada, pp. 278–287. ISBN: 9781605586359. DOI: 10.1145/1772954.1772993.
- [110] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. 2019. Initialize once, start fast: application initialization at build time. *Proc. ACM Program. Lang.*, 3, OOPSLA, Article 184, (October 2019), 29 pages. DOI: 10.1145/3360610.
- [111] Christian Wimmer, Codrut Stancu, David Kozak, and Thomas Würthinger. 2024. Scaling Type-Based Points-to Analysis with Saturation. *Proc. ACM Program. Lang.*, 8, PLDI, Article 187, (June 2024), 24 pages. DOI: 10.1145/3656417.
- [112] Christian Wimmer and Thomas Würthinger. 2012. Truffle: a self-optimizing runtime system. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH '12)*. Association for Computing Machinery, Tucson, Arizona, USA, pp. 13–14. ISBN: 9781450315630. DOI: 10.1145/2384716.2384723.
- [113] Niklaus Wirth. 1996. *Compiler Construction*. Addison-Wesley. ISBN: 978-0201403534.
- [114] Youfeng Wu and J.R. Larus. 1994. Static branch frequency and program profile analysis. In *Proceedings of MICRO-27. The 27th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 1–11. DOI: 10.1109/MICRO.1994.717399.

- [115] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013)*. Association for Computing Machinery, Indianapolis, Indiana, USA, pp. 187–204. ISBN: 9781450324724. DOI: 10.1145/2509578.2509581.
- [116] Haoran Xu and Fredrik Kjolstad. 2021. Copy-and-patch compilation: a fast compilation algorithm for high-level languages and bytecode. *Proc. ACM Program. Lang.*, 5, OOPSLA, Article 136, (October 2021), 30 pages. DOI: 10.1145/3485513.
- [117] Alina Yurenko. 2022. Revolutionizing Java with GraalVM Native Image — infoq.com. <https://www.infoq.com/articles/native-java-graalvm/>. [Accessed 20-02-2025]. (2022).
- [118] Fan Zhang, Naijie Gu, and Junjie Su. 2019. A Static Call Graph Construction Method Based on Simulation Execution. In *2019 IEEE 10th International Conference on Software Engineering and Service Science (ICSESS)*, pp. 72–76. DOI: 10.1109/ICSESS47205.2019.9040837.

