To Compile or Not To Compile: Evaluating Static Heuristics to Reduce Binary Size of Hybrid Execution Systems

Christoph Pichler
Johannes Kepler University
Linz, Austria
christoph.pichler@jku.at

Bernhard Urban-Forster
Oracle
Linz, Austria
bernhard.urban-forster@oracle.com

Paley Li
Everpure
Prague, Czech Republic
pali@purestorage.com

Roland Schatz
Oracle
Linz, Austria
roland.schatz@oracle.com

Stefan Marr
Johannes Kepler University
Linz, Austria
stefan.marr@jku.at

Abstract

To compile, or not to compile, that is the question: When 'tis nobler to optimize for performance. Modern compilers have many different optimizations and optimization goals. A common one is to balance peak performance and startup time. An ahead-of-time-compiled native executable that embeds a managed runtime may try to offer both, while reinforcing the notion that everything should be compiled. However, the cost of an enlarged binary size raises the question whether it is beneficial to compile everything.

In this paper, we evaluate static heuristics from classical AOT compilers as well as other techniques based on our own observations. Our goal is to identify heuristics that work in a compilation-first environment and that allow us to reduce binary size while maintaining peak performance.

We compare the different policies in a closed-world hybrid execution system for Java, based on GraalVM Native Image, on a set of 5 DaCapo and 13 Renaissance benchmarks. We find that with the best combination of heuristics we can reduce binary size by 20% while slowing down average performance by only 4%, but avoiding the need for any run-time feedback or complex machine-learning-based approaches. The most promising combination for production use combines heuristics based on early returns, estimated CPU cycles, number of parameters, and whether a method is a static initializer.

MPLR '26, Brussels, Belgium
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of 23rd ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '26)*, https://doi.org/XXXXXXX.XXXXXXX.

CCS Concepts: • **Software and its engineering → Runtime environments**; **Software performance**; *Source code generation*; *Interpreters.*

Keywords: AOT compilation, interpretation, performance

ACM Reference Format:
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 23rd ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '26).* ACM, New York, NY, USA, 14 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Modern runtime systems have long been shaped by the tension between two competing objectives: the desire for rapid startup and predictable deployment, and the pursuit of aggressive optimization and high steady-state performance. Ahead-of-time (AOT) compilation has traditionally been associated with the former, while just-in-time (JIT) compilation has been associated with the latter [4].

GraalVM Native Image [32] offers something different in this landscape. It produces a standalone native executable ahead of time, yet is a managed system, blurring the boundary between static and dynamic execution. It does not perform JIT compilation at run time. At the same time, it is not merely a traditional static compiler, as it preserves the semantic and run-time obligations of a managed environment.

These gains, however, are not without cost. They come in the form of greater pressure on closed-world analysis and configuration of dynamic features, and thus, producing binaries whose size reflects not only application code but also the inclusion of managed runtime support [24, 32].

Recent work began to tackle these issues: Hot compilation units can be highly optimized either just-in-time or ahead-of-time, while the cold code is executed in a less optimized version or even interpreted [10, 24]. This can lead to a small reduction of performance, but reduces binary size.

However, improvements in performance can only be gained if the highly optimized code is also executed often enough [4]. Identifying such hot code often relies on run-time information for profile-guided optimization, which introduces substantial time and energy overhead.

In this paper, we analyze the fop benchmark from the DaCapo suite [8] to identify code features and potential correlations to inform static heuristics. Then, we construct concrete heuristics, focusing on heuristics that are unlikely to increase compilation time. While being less precise than runtime data, static heuristics estimate hotness without having to run the application [3, 9, 10].

We implement the heuristics in the GraalVM compiler and use it in GraalVM Native Image, a hybrid execution system for JVM-based software, with a closed-world assumption.

We evaluate the heuristics with a subset[1] of the DaCapo [8] and Renaissance [28] benchmark suites. We assess the peak performance and the impact on the binary size.

We find that the most promising heuristic considers estimated CPU cycles per byte of code, number of parameters, early returns, and whether a method is a static initializer. With it we can reduce the binary size for Native Image executables for instance by 20%, while performance drops by only 4% on average (min. 3%, max. 10%).

Techniques such as profile-guided selection may yield a higher size reduction at a lower performance cost, but we believe the trade-offs to be still worthwhile to improve developer experience and for instance reduce cost, complexity, and turnaround time of CI pipelines.

Our main contributions in this work are:

- We describe code features and their correlations that apply to an AOT compiler-based hybrid execution system for Java (Section 3).
- We use the correlations to derive possible heuristics for reducing a Native Image executable by a dedicated size while keeping performance high (Section 4).
- We evaluate and compare heuristics across different benchmarks (Section 5).

2 Background

In this paper, we rely on the GraalVM Native Image hybrid execution system, which we introduce briefly, discussing its overall approach, execution modes, and implications for code size.

2.1 Hybrid Execution

We use the term *hybrid execution* to denote systems that combine ahead-of-time (AOT) execution with interpretation within a single runtime. Except for rare cases, the same code can be executed by any of these modes without requiring changes at source level. In particular, we exclude polyglot compositions such as native extensions from this definition, as they rely on rewriting code or mixing languages rather than switching execution modes.

[1]Not all benchmarks are supported for GraalVM Native Image with the experimental interpreter we used (see Section 5.1).

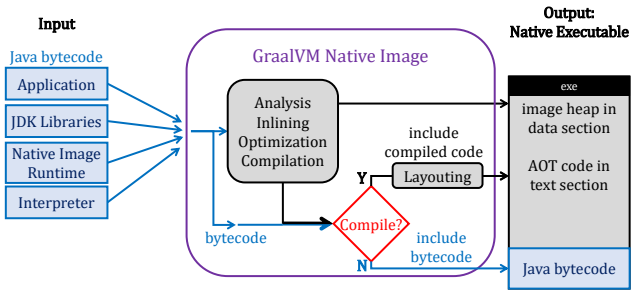

Figure 1. Native Image build process as described by Pichler et al. [24]

2.2 GraalVM and Native Image

GraalVM is a multi-language ecosystem [1, 35] based on OpenJDK [2]. It can use the aggressively and dynamically optimizing GraalVM compiler, which internally uses a graph-based "sea of nodes" intermediate representation called Graal IR [12]. This IR is used to perform optimizations such as inlining, escape analysis or loop optimizations. While execution can be based on a JVM (HotSpot™ VM as part of OpenJDK), there is also the possibility to create a standalone executable from Java bytecode, which is called GraalVM Native Image [32]. As an advantage in contrast to other execution modes of Java code, the standalone executable comes with a very fast startup, while aggressive optimizations still offer high peak performance.

During the build process, which is shown in Figure 1, Native Image takes the application and runtime code (libraries, JDK, and a minimal set of necessary runtime code for garbage collection and thread scheduling) and performs optimizations on it. Although Native Image performs application-wide static analysis including function inlining, compilation happens on a method level. Therefore, in this paper, *compilation units* in the context of Native Image correspond to methods including their inlined callees.

After compiling those compilation units, they are laid out so that caching effects help with fast startup [7]. At the end, the compiled compilation units are stored into a file following the determined layout, together with a heap snapshot.

Since Native Image operates in a closed-world setting, where no additional code is loaded at run time, a reachability analysis can be performed. However, because static analysis must balance scalability against precision, causing some unreachable code still to be included in the final executable [33]. As a result, the compiled artifact can be substantially larger than the corresponding Java class files loaded dynamically by the JVM [24].

2.3 Hybrid Execution within GraalVM Native Image

Pichler et al. [24] proposed hybrid execution within GraalVM Native Image: While hot code is compiled to machine code ahead-of-time, cold code is kept as compact Java bytecode. When called, cold code can be executed with a lightweight interpreter. Therefore, heuristics have to decide the execution mode for every compilation unit, as shown in Figure 1: For every compilation unit, depending on the decision, either AOT-compiled machine code, or Java bytecode is included into the final executable. While code is currently compiled before this decision is taken, we discuss possible future reductions of compilation effort in Section 7.

As single wrong decisions can strongly impact performance negatively, the compiled code can replace the Java bytecode at run time if a compilation unit is called twice within a certain time range. In our experiments, only $\approx 2\%$ of all interpreted units are replaced again. In future, this could be done by a simple JIT compiler instead [31].

Existing hybrid systems such as the Java VM or approaches that cache JIT-compiled results for future runs [5, 19, 21, 23] have in common that they eventually trigger compilation for all used code, whether ahead-of-time or at run time. However, Pichler et al. [24] have shown that it can be worthwhile not to compile everything.

2.4 Code Size

Including unreachable (but not proven to be unreachable) code into the compilation result, as described in Section 2.2, is not the only reason for large binary executables: Also, due to inlining, branch duplication, etc., even code of a single compilation unit is often larger than a less-optimized version [15, 16, 18, 27, 29]. Optimizing for peak performance results in larger binaries, while optimizing for smaller binaries usually comes with lower peak performance. Thus, compilers such as gcc [26] or clang [14] offer different kinds of optimization levels.

As an example for GraalVM Native Image, AOT-compiled machine code of the DaCapo benchmark suite [8] is 3.1× larger than its original Java bytecode. The distribution of the DaCapo benchmarks where every dot represents one compilation unit is shown in Figure 2. Therefore, also GraalVM Native Image produces smaller executable files when less code is compiled ahead of time [24].

3 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 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.

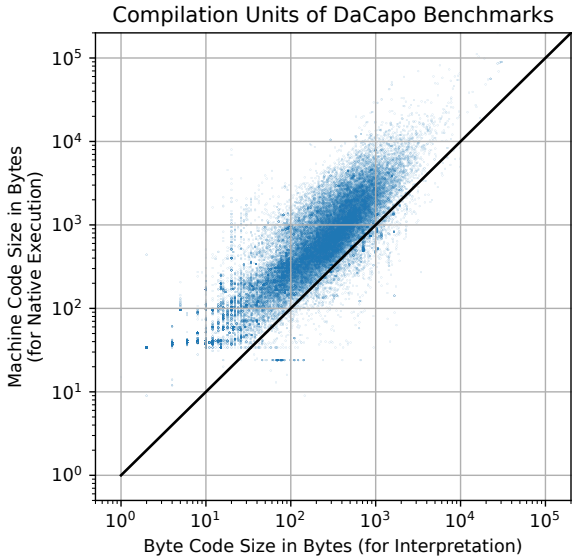

Figure 2. Comparison: Java Bytecode vs. corresponding Machine Code for Compilation Units in DaCapo [8]

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

3.1 Methodology

In the analysis, we focus for most parts on the largest DaCapo [8] 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.

We extract the data for our analyses during the build process of a GraalVM Native Image executable. As shown in Figure 1, the GraalVM 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 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 4.

- *nCycles*: number of estimated CPU cycles, estimated by a static heuristic [17, 20].
- *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 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 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` [30], 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, the self time of a compilation unit is measured: Time of inlined methods, thus part of the same compilation unit, is included. Time of non-inlined callees, thus part of other compilation units, is not included. Compilation units without activations are excluded from this analysis.

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 [8] *fop* benchmark is shown in Figure 3.

In contrast to the static features listed above, this 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.

3.3 Correlations of Static Features and the Execution Time Ratio

For the following sections, we analyze how static features related to the execution ratio of Section 3.2. To illustrate correlations, we include scatter plots, where every dot represents a compilation unit. While the x axis represents the static feature, the y axis indicates the execution time ratio. We also provide Pearson's and Spearman's correlation coefficients, which both provide a unitless number: −1 denotes a perfect inverse, linear correlation, +1 a perfect positive linear correlation. 0 indicates that there is no correlation. While Pearson's coefficient indicates linear correlation, Spearman's coefficient works on ranked data. They are computed as shown in Equations (1) and (2), respectively.

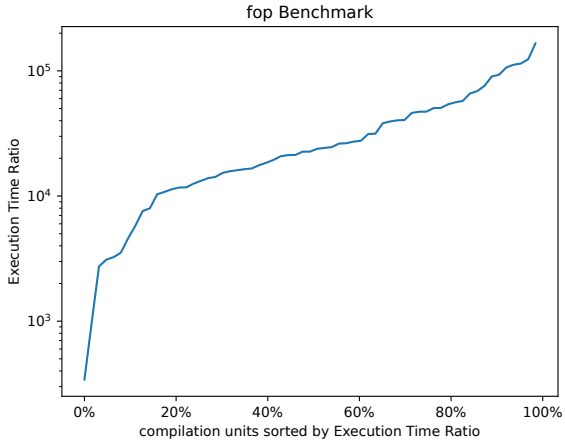

Figure 3. Distribution of the ratio $\frac{\text{interpreter time}}{\text{AOT execution time}}$ for compilation units of the DaCapo *fop* benchmark. It includes all compilation units activated at least once. It shows that for more than 90%, the benefit of AOT compilation is larger than a factor 400.

$$\rho_{Pearson}(X, Y) = \frac{cov(X, Y)}{\sigma_X \cdot \sigma_Y} \tag{1}$$

$$\rho_{Spearman}(X, Y) = \rho_{Pearson}(R[X], R[Y]) \tag{2}$$

3.3.1 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. This is inspired by Pichler et al. [24], who used a similar heuristic, but included run-time information.

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 Figure 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).

3.3.2 Number of Parameters. A feature that the machine-learning-based approach of Cugurovic et al. [10] identified as potentially relevant is the number of input parameters of a compilation unit. We do not consider parameters of inlined callees, since they have been replaced by local variables within the compilation unit. One may speculate that there is a deeper correlation between number of parameters and complexity of a method. Figure 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

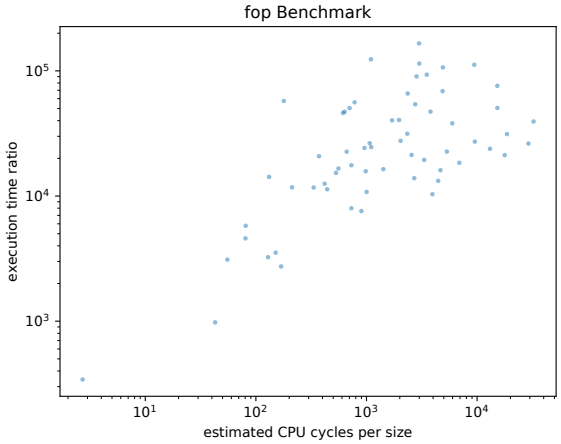

Figure 4. Estimated computational density of cycles per byte compared to the execution time ratio. Higher density, i.e., more estimated CPU cycles per size benefits more from AOT compilation.

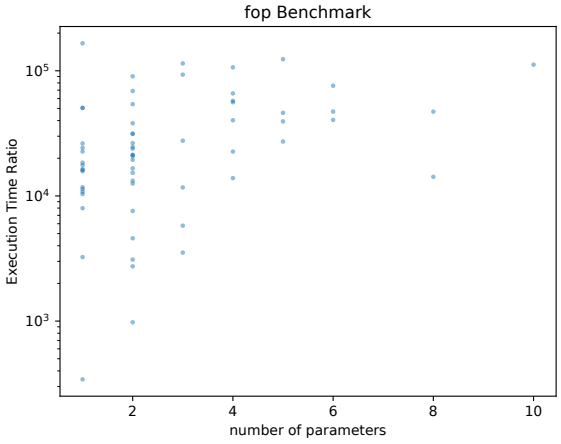

Figure 5. The number of parameters of a compilation unit compared to the execution time ratio. The more parameters a compilation unit has, the more compilation pays off.

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. The correlation coefficients are +0.38 (Pearson) and +0.41 (Spearman).

3.3.3 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 6. Every line represents a compilation unit, the x value denotes the number of the activation. The y

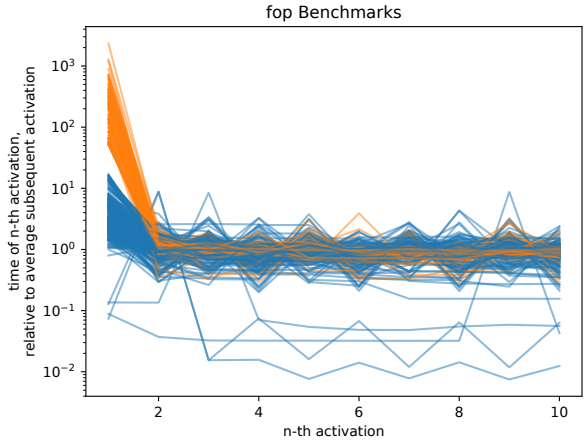

Figure 6. Activation Times of Compilation Units: There are some compilation units (marked orange) whose first activation takes much longer than subsequent ones. The y value is computed as in Equation 3.

value is the relative time of a single activation w.r.t. the average time of all activations but the first one; it is computed as in Equation 3 ($time(n)$ denotes the time of the n^{th} activation, N is the total number of activations).

$$\text{relative time of } n^{th} \text{ activation} = \frac{time(n)}{\frac{1}{N-1} \cdot \sum_{i=2}^{N} time(i)} \quad (3)$$

While a first execution usually takes more time for initialization, there is also a group of compilation units, marked orange in Figure 6, where their first activation takes much longer than for other compilation units (marked blue).

This overhead can have different reasons. We hypothesized that one of them is that these compilation units perform some form of initialization during the first activation.

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 $\frac{minRetDist}{maxRetDist}$. 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 Figure 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.

3.3.4 Pointer `Null` Checks. Inspired by the early returns, we also investigated pointer comparisons with `null`, which are also used in the literature [9, 34]. The idea here is that

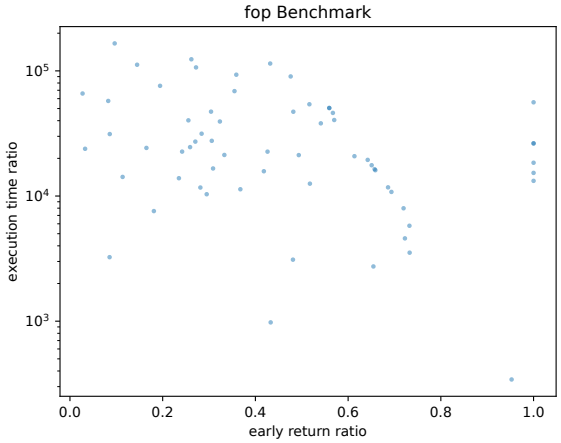

Figure 7. Scatter plot showing the early return ratio compare to the execution time ratio. Although the data is rather scattered, there is a light indication (correlation coefficient −0.33) that compilation might pay off in cases of early returns.

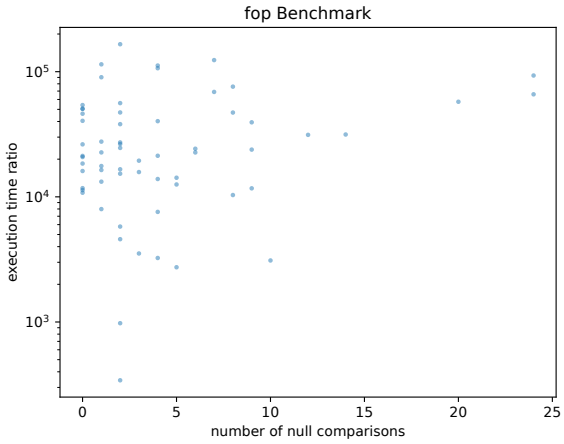

Figure 8. Scatter plot showing the number of `null` checks compare to the execution time ration. More `null` checks seem to indicate a higher benefit of compilation for compilation unit.

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 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.

3.4 Static Initializers

Static initializers in Java are methods for class-level data initialization. For most static initializers, our measurements

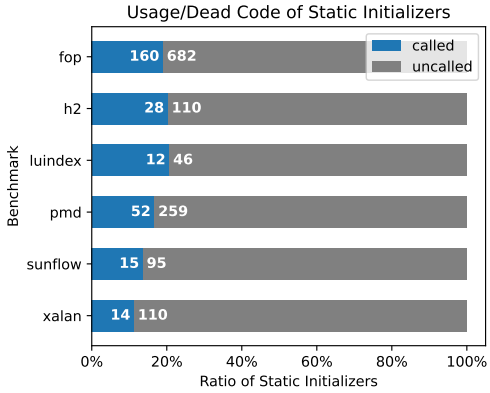

Figure 9. Usages of Static Initializers in DaCapo Benchmarks

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.

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 [8] benchmark suite, Figure 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 [33] as mentioned in Section 2.2. 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 example, Figure 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

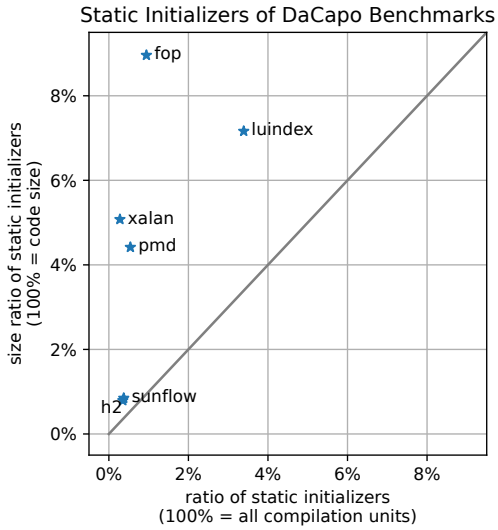

Figure 10. Sizes of Static Initializers in DaCapo Benchmarks.

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 ≈ 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.

4 Static Heuristics

Based on the analyses in Section 3, here we formulate static 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 3.1, the decision 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.

4.1 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. Similarly, a value of ∞ indicates that it has to be compiled, which is true for some compilation units of the runtime system, that are not supported to be interpreted. This includes for instance the interpreter itself or core parts of the Native Image runtime [24, 32].

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 3.1), 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 2.2).

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

1. Receive the list of compilation units during AOT compilation, as described in Section 3.1.
2. Sort the list by the chosen heuristic score function.
3. Set `goalSize` to $\texttt{baseSize} \cdot (1 - \texttt{sizeRed})$.
4. Set `curSize` to `baseSize`.
5. While `curSize > goalSize`:
   a. Take the next candidate for interpretation (i.e., the compilation unit with the lowest score in the list), and set its execution mode to interpretation.
   b. 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.

4.2 Single-feature Heuristics

Based on the analysis in Section 3, 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 1.

When using single-feature heuristics, normalization does not alter the ranking of compilation units. However, normalization becomes critical when combining multiple features, as it influences relative impacts of single features. Thus, for consistency, we normalize all single-feature heuristic scores upfront, even when only one single feature is considered.

Estimated CPU cycles per byte: `ecyc` (based on Section 3.3.1). The positive correlation we saw indicates to us that more computationally dense compilation units cause a higher overhead when they are interpreted.

To smooth 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 3.3.2). While the correlation is not as clear as for `ecyc`, a higher number of parameters for a compilation unit seems to indicates 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: `rtrn` (based on Section 3.3.3). For the early returns, we found 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 heuristic.

`Null` ***Pointer Checks:*** `null` (based on Section 3.3.4). Similar to early returns, we noticed that compilation units with many `null` checks may benefit from compilation. Thus, we determine the heuristic score by taking the number of `null` comparisons and increment it by one, $nNullCmp + 1$.

Static Initializer: `init` (based on Section 3.4). 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`.

4.3 Combined Heuristics

Combining single features into more complex heuristics often leads to better results [10, 34].

We combine a set of heuristics `H` into a new heuristic by multiplying the single score values as in Equation 4. 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.

$$heur_H(cu) = \prod_{h \in H} heur_h(cu) \tag{4}$$

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 5.

$$\begin{aligned} heur_{iprn}(cu) =& heur_i(cu) \cdot heur_p(cu) \cdot heur_r(cu) \cdot heur_n(cu) \\ =& init(cu) \cdot (nPars + 1) \cdot \\ & \frac{maxRetDist}{minRetDist} \cdot (nNullCmp + 1) \end{aligned} \tag{5}$$

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 5).

5 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.

5.1 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. GraalVM's Profile-guided optimization (PGO) [22] was not used, neither for retrieving metrics of single heuristics such as `ecyc`, nor for benchmark execution. 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.

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.

Table 1. 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	$cu \rightarrow \begin{cases} 0: & \text{cu is static initializer} \\ 1: & else \end{cases}$
`pars`	`p`	number of parameters	$cu \rightarrow nPars + 1$
`null`	`n`	number of pointer comparisons against `null` [34]	$cu \rightarrow nNullCmp + 1$
`rndX`	`-`	random heuristic	$cu \rightarrow rnd(1.0, 10.0)$, fixed seed
`rtrn`	`r`	inverse early return ratio	$cu \rightarrow \frac{maxRetDist}{minRetDist}$

We assess the single-feature heuristics `ecyc`, `pars`, `rtrn`, and `null` (see Table 1). 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 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 1.

Interpreter Size and Execution Mode. As described, our hybrid execution replaces AOT-compiled code by Java bytecode, as shown in Figure 1. 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 AOT-compiled code replaces the Java bytecode at run time if a compilation unit is called twice within 100ms. In our experiments, only ≈ 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.

Illustration. We use box plots to illustrate the results. The box extends from the first to the third quartile (i.e., the inter-quartile 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× the inter-quartile range from the box. Circles denote data points outside of the whiskers.

5.2 Results for Single-Feature Heuristics

Figure 11 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 ≈ 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.

5.3 Results for Combined Heuristics

The results for combined heuristics are shown in Figure 12. 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 performance increases are 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`.

5.3.1 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.

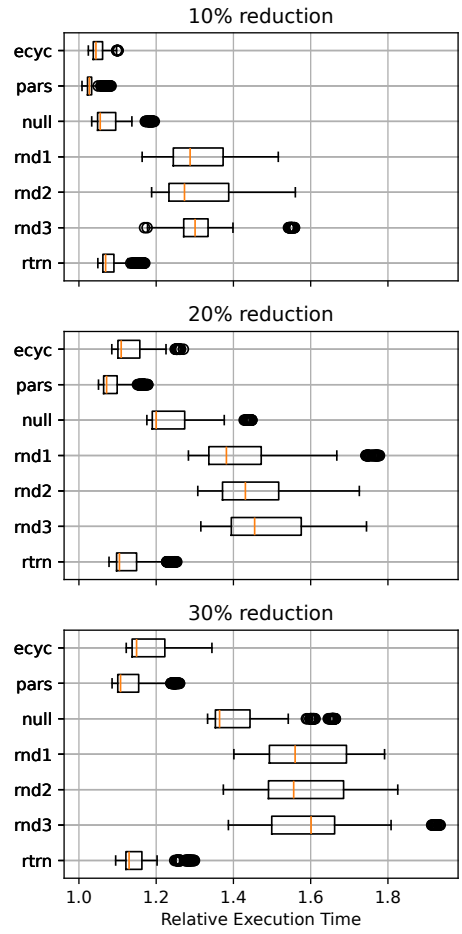

Figure 11. Single-feature results, grouped by size reduction and heuristics. The heuristic based on number of parameters (pars) achieves the size reduction with the lowest slowdown, and thus, is the best individual heuristic.

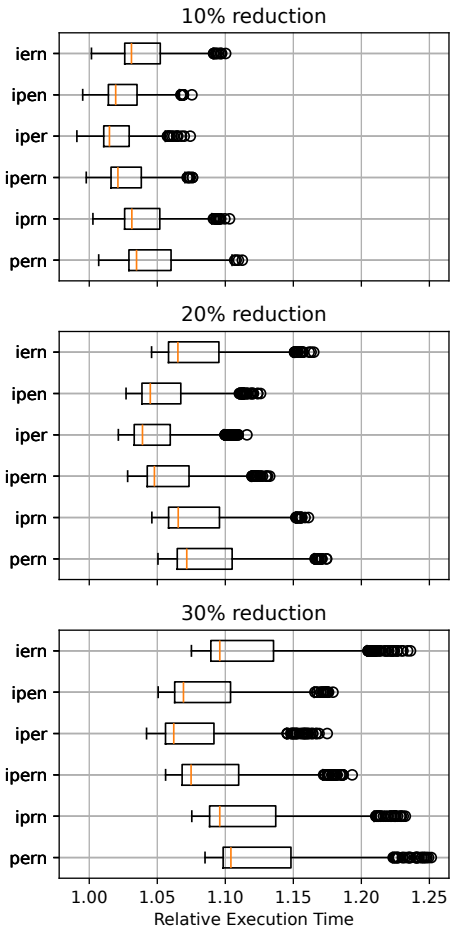

Figure 12. Combined results, grouped by size reduction and heuristic. Overall, the `iper` combination of heuristics achieves the size reduction with the lowest overhead.

Figure 13 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 14 plots the medians for the size reduction targets between 10% and 35%. Here we also include the random heuristics, though, they drop off the 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.

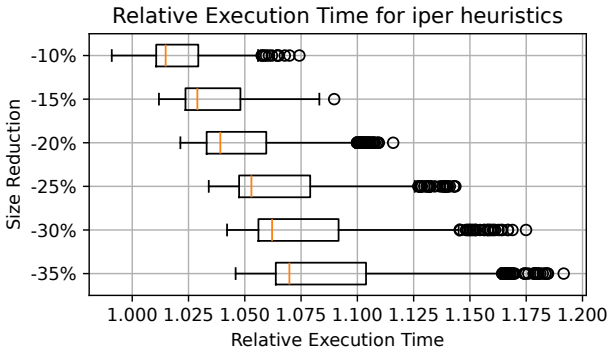

Figure 13. Trade-off between size and performance for our `iper` heuristics. At a size reduction of 35% we see a worst case slowdown of less than 20% with the median slowdown staying under 7%.

5.3.2 Impact on the Individual Benchmarks. As the box plots already indicated, the impact on individual benchmarks can differ significantly. Thus, Figure 15 shows the

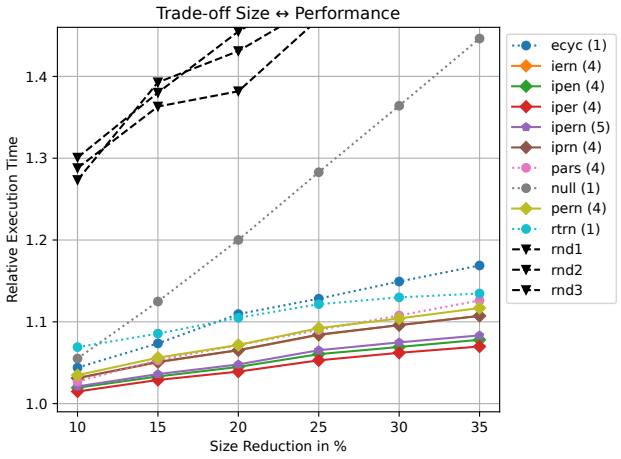

Figure 14. Trade-off between size and performance per heuristic. Showing the median slowdown. `iper` is the overall best candidate, while the random heuristics are causing the highest slowdowns overall.

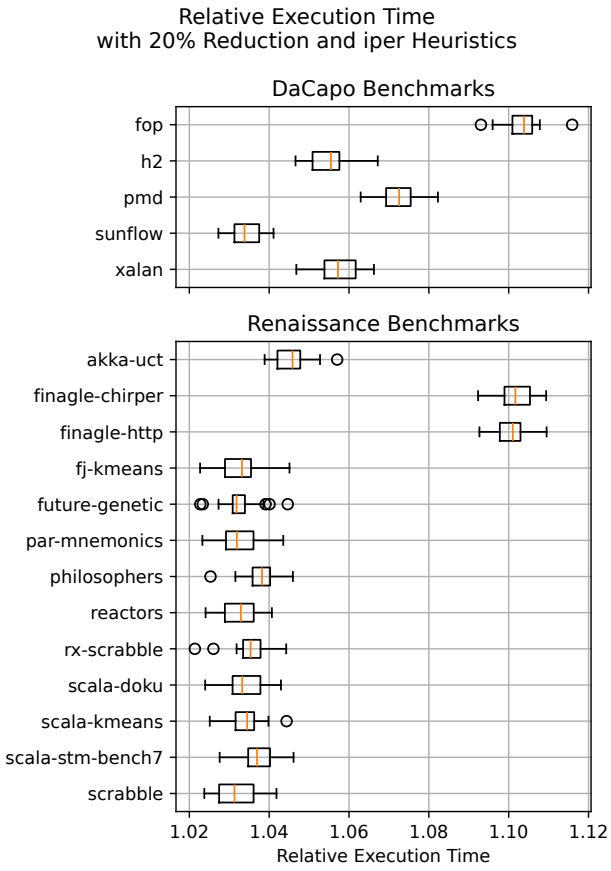

Figure 15. Detailed benchmark results for a 20% size reduction using the `iper` heuristic. The highest slowdown is caused on fop, while most Renaissance benchmarks see only minor slowdowns.

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 to identify candidate heuristics. We consider it a good indication that we did not overfit to the benchmark we looked at for correlations.

5.4 System Idiosyncrasies

The observed effectiveness of the different heuristics may depend on implementation-specific behaviors of the lightweight interpreter used. For example, inefficient parameter handling could be a reason why compilation units with many parameters are more likely candidates for compilation. While this limitation can introduce a bias in our evaluation, it does not invalidate the usability of the heuristics. Rather, it shows a need for further validation across different systems. Similarly, the `ecyc` score relies on static estimations, which may not be available or equally accurate in all ecosystems. In contrast, the impact of initializers is likely transferable to other systems, as it originates more from language semantics rather than from implementation-specific reasons.

6 Related Work

The first compiler heuristics may have been proposed all the way back by Backus et al. [6]. Since then, the field has flourished and we briefly review approaches to static predictions and profile-guided approaches similar to ours.

Static Prediction. Approaches to statically predict program behavior are typically created similarly to our methodology here. First, they collect data on code features and performance to identify suitable heuristics that enable a somewhat reliable prediction of program behavior at compilation time and thereby guide compilation decisions. For example, Calder et al. [9] guide optimizations such as inlining, branch prediction, or loop unrolling based on static predictions. Debray et al. [11] perform optimizations to achieve more compact code. However, our scenario as described in Section 2.3 differs, since the GraalVM compiler already performs 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 [13, 25]. 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.

Profile-Guided Compilation. Pichler et al. [24] proposed hybrid execution model to reduce binary size for GraalVM Native Image executables. Their approach relies on run-time information that they collected in additional profiling runs. With the profiling information, they can achieve a reduction of the overall binary size of 36% without significantly impacting performance. However, the profiling runs are time-intensive and require additional re-compilation, adding to complexity and cost of the build infrastructure.

A static approach of a per-method based hotness prediction has been presented recently by Cugurovic et al. [10]: GraalMHC uses a machine learning model to decide the optimization level per compilation unit for GraalVM Native Image. However, although ML-based approaches do not rely on run-time information, they come with high training costs.

Another approach is caching JIT-compiled code [5, 19, 23], which also reduces compilation effort and balances startup and peak performance. However, while this reduces the size of the initial binary, it has its own tradeoffs. The first iterations, that do not yet have JIT-compiled code available, still have the warm-up cost. While subsequent runs can then benefit from the cache, it requires additional storage and complexity, too.

7 Future Work

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 currently compile all compilation units during the build process, as described in Section 3.1. 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 also to 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.

As the proposed hybrid system currently only supports AOT compilation and interpretation, a JIT compiler could reduce the slowdown that interpreted compilation units cause in the rare case they are relevant for peak performance in practice. As previously shown by Pichler et al. [24], less than 5% of the interpreted methods in the DaCapo benchmarks would be candidates for JIT compilation, and even less, 2%, in our experiments. A JIT compiler would also give us the flexibility to exclude large compilation units from the image to reduce binary size further without reducing peak performance. However, this would come at the cost of reintroducing a warmup phase.

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 while requiring less training data or using few run-time profiling runs. Arnold et al. [3] previously showed how to efficiently combine such static and dynamic information for inlining.

8 Conclusions

In this paper, we evaluated how different static code features correlate with the ratio between execution time of AOT-compiled compilation units and interpreted compilation units. We did this work with a closed-world hybrid execution system based on GraalVM Native Image.

We evaluated five static heuristics based on the analyzed correlations and evaluated them with respect to binary size reduction and their effects on peak performance. Specifically, we looked at a heuristic based on estimated CPU cycles per byte of compiled code, at number of parameters, an early return ratio, number of `null` pointer checks, and whether a compilation unit is a static initializer.

We evaluate the heuristics on the 5 DaCapo and 13 Renaissance benchmarks that are currently supported by Native Image and our experimental interpreter. The most promising single-feature heuristic is the number of parameters. However, the best overall heuristic is the combination of estimated CPU cycles, number of parameters, early returns, and whether a method is a static initializer. This one allows us to reduce the binary size by 20% at an average slowdown of only 4%. Depending on the size reduction one aims for, one can further tune how much slowdown is acceptable. When reducing size by 10% or 30%, we see a performance slowdown of 2% or 6% respectively.

Acknowledgments

This research project was partially funded by Oracle.

References

[1] 2025. GraalVM — graalvm.org. https://www.graalvm.org/. [Accessed 20-02-2025].

[2] 2025. OpenJDK — openjdk.org. https://openjdk.org/. [Accessed 04-02-2026].

[3] Matthew Arnold, Stephen Fink, Vivek Sarkar, and Peter F. Sweeney. 2000. A comparative study of static and profile-based heuristics for inlining. *SIGPLAN Not.* 35, 7 (Jan. 2000), 52–64. doi:10.1145/351403.351416

[4] John Aycock. 2003. A brief history of just-in-time. *ACM Comput. Surv.* 35, 2 (June 2003), 97–113. doi:10.1145/857076.857077

[5] 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)*. 481–495. doi:10.1109/EuroSP.2017.43

[6] 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* (Los Angeles, California) *(IRE-AIEE-ACM '57 (Western))*. ACM, 188–198. doi:10.1145/1455567.1455599

[7] 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* (Las Vegas, NV, USA) *(CGO '25)*. Association for Computing Machinery, New York, NY, USA, 689–703. doi:10.1145/3696443.3708927

[8] 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. *SIGPLAN Not.* 41, 10 (Oct. 2006), 169–190. doi:10.1145/1167515.1167488

[9] 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 (Jan. 1997), 188–222. doi:10.1145/239912.239923

[10] 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* (Sydney, NSW, Australia) *(CC '26)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3771775.3786276

[11] 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. doi:10.1145/349214.349233

[12] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages* (Indianapolis, Indiana, USA) *(VMIL '13)*. ACM, 1–10. doi:10.1145/2542142.2542143

[13] 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*. 187–194. doi:10.1109/SCCC.2011.25

[14] LLVM Foundation. 2026. Clang: a C language fammily frontend for LLVM. https://clang.llvm.org/. [Accessed 06-02-2026].

[15] Arvind Krishnaswamy and Rajiv Gupta. 2002. Profile guided selection of ARM and thumb instructions. *SIGPLAN Not.* 37, 7 (June 2002), 56–64. doi:10.1145/566225.513840

[16] Sheayun Lee, Jaejin Lee, Chang Yun Park, and Sang Lyul Min. 2004. A Flexible Tradeoff Between Code Size and WCET Using a Dual Instruction Set Processor. In *Software and Compilers for Embedded Systems*, Henk Schepers (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 244–258.

[17] 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* (Boston, MA, USA) *(VMIL 2018)*. Association for Computing Machinery, New York, NY, USA, 26–35. doi:10.1145/3281287.3281290

[18] Paul Lokuciejewski, Sascha Plazar, Heiko Falk, Peter Marwedel, and Lothar Thiele. 2011. Approximating Pareto optimal compiler optimization sequences—a trade-off between WCET, ACET and code size. *Software: Practice and Experience* 41, 12 (2011), 1437–1458. doi:10.1002/spe.1079 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.1079

[19] 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 (Oct. 2023), 22 pages. doi:10.1145/3622839

[20] 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* (Las Vegas, NV, USA) *(CGO '25)*. Association for Computing Machinery, New York, NY, USA, 123–136. doi:10.1145/3696443.3708958

[21] OpenJDK. 2026. Project Leyden. https://openjdk.org/projects/leyden/. [Accessed 05-02-2026].

[22] 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 20-04-2026].

[23] 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* (Pasadena, CA, USA) *(VMIL '24)*. Association for Computing Machinery, New York, NY, USA, 43–53. doi:10.1145/3689490.3690406

[24] 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* (Singapore, Singapore) *(MPLR '25)*. Association for Computing Machinery, New York, NY, USA, 12–22. doi:10.1145/3759426.3760976

[25] 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.* 494–501. doi:10.1109/MASCOT.2004.1348305

[26] GNU Project. 2026. GCC — the GNU Compiler Collection. https://gcc.gnu.org/. [Accessed 06-02-2026].

[27] Aleksandar Prokopec, Gilles Duboscq, David Leopoldseder, and Thomas Würthinger. 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)*. 164–179. doi:10.1109/CGO.2019.8661171

[28] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: benchmarking suite for parallel applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming*

Language Design and Implementation (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 31–47. doi:10.1145/3314221.3314637

[29] Robert W. Scheifler. 1977. An analysis of inline substitution for a structured programming language. *Commun. ACM* 20, 9 (Sept. 1977), 647–654. doi:10.1145/359810.359830

[30] L. Torvalds. 2026. Linux Kernel. https://github.com/torvalds/linux/tree/master/tools/perf. [Accessed 11-03-2026].

[31] Oracle via GitHub. 2026. Project Crema – Open the World for Native Image. https://github.com/oracle/graal/issues/11327. [Accessed 05-02-2026].

[32] 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 (Oct. 2019), 29 pages. doi:10.1145/3360610

[33] 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

[34] 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*. 1–11. doi:10.1109/MICRO.1994.717399

[35] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Indianapolis, Indiana, USA) *(Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 187–204. doi:10.1145/2509578.2509581