Exploiting Partially Context-sensitive Profiles to Improve Performance of Hot Code

MAJA VUKASOVIC, School of Electrical Engineering, University of Belgrade, Serbia
ALEKSANDAR PROKOPEC, Oracle Labs, Switzerland

Availability of profiling information is a major advantage of just-in-time (JIT) compilation. Profiles guide the compilation order and optimizations, thus substantially improving program performance. Ahead-of-time (AOT) compilation can also utilize profiles, obtained during separate profiling runs of the programs. Profiles can be context-sensitive, i.e., each profile entry is associated with a call-stack. To ease profile collection and reduce overheads, many systems collect partially context-sensitive profiles, which record only a call-stack suffix. Despite prior related work, partially context-sensitive profiles have the potential to further improve compiler optimizations.

In this article, we describe a novel technique that exploits partially context-sensitive profiles to determine which portions of code are hot and compile them with additional compilation budget. This technique is applicable to most AOT compilers that can access partially context-sensitive profiles, and its goal is to improve program performance without significantly increasing code size. The technique relies on a new hot-code-detection algorithm to reconstruct hot regions based on the partial profiles. The compilation ordering and the inlining of the compiler are modified to exploit the information about the hot code. We formally describe the proposed algorithm and its heuristics and then describe our implementation inside GraalVM Native Image, a state-of-the-art AOT compiler for Java. Evaluation of the proposed technique on 16 benchmarks from DaCapo, Scalabench, and Renaissance suites shows a performance improvement between 22% and 40% on 4 benchmarks, and between 2.5% and 10% on 5 benchmarks. Code-size increase ranges from 0.8%–9%, where 10 benchmarks exhibit an increase of less than 2.5%.

CCS Concepts: • **Software and its engineering** → **Compilers**; *Runtime environments;*

Additional Key Words and Phrases: Ahead-of-time compilation, inlining, inline substitution

ACM Reference format:
Maja Vukasovic and Aleksandar Prokopec. 2023. Exploiting Partially Context-sensitive Profiles to Improve Performance of Hot Code. *ACM Trans. Program. Lang. Syst.* 45, 4, Article 20 (November 2023), 64 pages.
https://doi.org/10.1145/3612937

1 INTRODUCTION

Just-in-time (JIT) compilation is performed online during the execution of the program and is done on a subset of frequently executed methods—the other parts of the code are executed by an interpreter [38]. **Ahead-of-time (AOT)** compilation is an alternative approach in which the

Authors' addresses: M. Vukasovic, School of Electrical Engineering, University of Belgrade, Belgrade, Serbia, 11000; e-mail: maja.vukasovic@etf.bg.ac.rs; A. Prokopec, Oracle Labs, Zurich, Switzerland, 8004; e-mail: aleksandar.prokopec@oracle.com.

0164-0925/2023/11-ART20 $15.00
https://doi.org/10.1145/3612937

program is compiled to the target machine code before its execution begins. AOT compilation overcomes one of the main issues of JIT—programs are slow during start-up due to being interpreted before getting compiled [44]. However, **profile-guided optimizations (PGO)** are the hallmark of JIT compilers: During the initial interpretation of the program, the runtime environment collects profile information about the program's execution, such as the frequency of conditionals or the object types seen at different program locations, and relays this information to the optimizing compiler once the compilation starts. This can significantly improve the effectiveness of many compiler optimizations [20, 29, 30, 34, 55, 56, 72, 76, 78, 97, 103].

Nevertheless, profile-driven optimizations are not restricted to JIT compilation—in AOT compilation, the lack of runtime profiling is mitigated with offline profiling runs [68, 85, 110]. One common approach to offline profiling is to generate an instrumented program binary, which collects certain information about the program's execution [20, 110, 120]. The information collected by the instrumented binary is then provided back to the AOT compiler, which uses the profiles to create a second, optimized binary, with the aim of improving program performance.

Profiling information can be broadly categorized as context-insensitive and context-sensitive: the former associates each code location with at most one profile entry, while the latter associates each code location to multiple entries, each of which corresponds to a *context* in which the respective code location was executed. In this article, the contexts are the methods on the stack at the time when the profile was collected—in other words, call stacks (which is usually what the term "context-sensitive" more narrowly implies). While most JIT-based runtime environments collect context-insensitive profile information for reasons of simplicity and low overhead [15, 23, 50, 80, 122], offline profiling facilitates the collection of accurate context-sensitive profiles [42]. Still, accurate collection of context-sensitive profiles can be expensive even in an offline setting (each profile entry needs to be associated with the complete call stack), so multiple approaches were developed to reduce the profiling overheads [35, 41, 47, 77, 119]. One common approach to reducing these overheads is to collect *partially context-sensitive profiles*, which associate each profile entry with only a conveniently selected suffix of the call stack [31, 37, 101, 106, 120]. While the collection of partially context-sensitive profiles can be as efficient as context-insensitive collection, it is not clear how to best utilize partially context-sensitive profile information to aid compiler optimizations.

This article presents an algorithm for utilizing partially context-sensitive profiles to improve compilation. The algorithm uses the partial profiles to decide which larger parts of the code are "hot" and then forms compilation units that "cover" those "hot" parts. Its goal is to *improve performance* of the generated code, *without significantly increasing the size* of the program binaries.

The main idea in the proposed algorithm is to exploit partially context-sensitive profiles to opportunistically reconstruct hot portions of the code and to improve inlining in those parts of the code. To achieve this, the algorithm modifies the compilation order and the inlining decisions of the compiler: Throughout the text, we therefore refer to it as a *compilation-scheduling and inlining algorithm*. In the first phase, the algorithm "stitches" the partially context-sensitive profiles to form a forest of *breadcrumb trails*—trees that trace parts of the program that are frequently executed. In the second phase, the algorithm starts the compilation from the roots of these hotness trails and compiles them with more aggressive inlining decisions. The remaining cold portions of the code are thereafter compiled as they would be without the modifications.

Contributions. After summarizing the problem in Section 2, we first explain the algorithm formally and then on the implementation level. The main contributions in this article are as follows:

- A novel compilation technique that uses partially context-sensitive profiles to improve the performance of ahead-of-time compiled programs, formalized in Section 3. This technique comprises an algorithm that opportunistically detects hot code using partial contexts, as

explained in Sections 3.3 and 3.4, and modifications to existing inlining and compilation scheduling, as explained in Sections 3.5 and 3.6.

— A complete, production-ready implementation of the proposed technique in GraalVM's ahead-of-time compiler called Native Image, described in detail in Section 4. Concretely, within the existing GraalVM compiler, we modified the compilation queue, added a new phase for the profile analysis, and used it to augment the existing inliner.

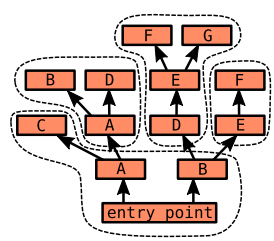
— An evaluation on 16 benchmarks from DaCapo [46], Scalabench [108], and Renaissance [99] benchmarking suites, presented in Section 5. Compared to the previous implementation of the profile-guided optimizations in the Enterprise version of Native Image, the proposed algorithm improves performance in the range of 22%–40% on 4 benchmarks, and 2.5%–10% range on 5 benchmarks. Our approach increases the compiled-code size between 0.8% and 9%; for 10 out of 16 benchmarks the compiled-code size is increased only up to 2.5%. We also compared the introduced changes against GraalVM Native Image without PGO, standard GraalVM compiler in JIT mode, and HotSpot's C2 compiler in JIT mode. After the main results, the evaluation shows how the algorithm parameters were tuned for performance and presents a breakdown of performance-contributing factors.

In Section 7, we present an overview of related work in the area of profiling, profile-guided optimizations, inlining and ahead-of-time compilation, and we conclude the article in Section 8.

2 PROBLEM STATEMENT

In this article, we address the problem of utilizing accurate partially context-sensitive execution profiles to improve *compilation-scheduling and inlining decisions* in a way that *decreases the running time* of compiled programs *without significantly increasing the binary size*. Concretely, the input to the problem is a set of profile entries, where each entry consists of a partial calling context and the execution count associated with that calling context. Here, *profile entries* include at least those program locations that are conditional branches and virtual (indirect) calls. Each entry includes the execution count (for conditionals) or a mapping from the receiver-type to the number of occurrences of that type (for virtual calls). The term *partial* means that calling contexts do not consist of all the procedure calls starting from the program entry point and up to the profiled location, but only of some suffix thereof. By *accurate*, we mean that there exists a profile entry for each profiled program location that was executed during the profiling run and that the respective execution counts are exact (not approximate).

Furthermore, an important assumption that we make on the input is that the set of profile entries forms a *compilation-unit-wise partition*. To explain this, consider the adjacent figure that shows an example activation tree. An activation tree is a tree that contains all call stacks that exist during some execution of the program (this is formally defined in Section 3). Each node in the activation tree represents one subroutine of the program—for example, the program `entry point` subroutine is the root of the tree, while the subroutines `A` and `B` are invoked from the `entry point` subroutine.



Activation tree and its compilation units

A compilation unit consists of a root subroutine, along with a connected set of subroutines that were inlined into the root subroutine. In the figure, compilation units are delineated with a dashed line. A set of profile entries forms a compilation-unit-wise partition if (1) each subroutine in the compiled program on which the profiles were obtained is a root of at most one compilation unit, (2) each profile entry has a calling context that corresponds to some compilation unit (i.e.,

starts the subroutine that is the root of the compilation unit and contains only subroutines within that compilation unit).

In the example, there are four compilation units: one rooted at `entry point`, one at `A`, one at `D`, and another at `E`. The calling contexts `entry point→A→C` and `A→D` are valid calling contexts in a compilation-unit-wise partition, because each of them starts with the compilation-unit root and is completely within that compilation unit. The context `E→G` is not valid in that compilation-unit-wise partition, because it does not start with a compilation-unit root. The context `entry point→B→D` is also not valid, because it is not confined within a single compilation unit. Our assumption is that the input profile contains only valid entries for some compilation-unit-wise partition.

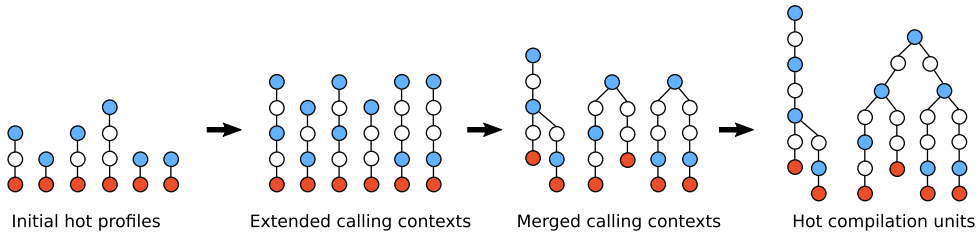
Rationale. The constraints imposed on the input profiles may seem artificial at first glance, but they are a natural consequence of performing instrumentation within compilation units. Each position in a compilation unit (which consists of inlined subroutines) is assigned a set of unique counters, each corresponding to a calling context *within* that compilation unit. The GraalVM Native Image, in which we implemented the proposed algorithm, collects accurate, partially context-sensitive conditional and receiver-type profiles, which form a compilation-unit-wise partition. Native Image does so by producing an instrumented binary of the program, which is used to collect the profiles. The optimized binary is then created using these profiles. This two-tier approach is common: similar instrumented binaries can be created by GCC [20], LLVM [16], and Scala Native [110].

AOT compilation of a program assumes that all of the program's methods have been compiled before the execution of the program, with the exception of those methods that are inlined at all callsites. Most of the methods contain one or more callsites, i.e., locations from which other methods are invoked (callees). During a single compilation of a method, a decision is made whether a callee at each callsite should be inlined or not. If the callee method is inlined, then its body replaces the invocation at a callsite and transitively its callees are considered for inlining. The callees that remain non-inlined are left to be compiled later in the compilation process. In this setting, a method is usually compiled only once. Since the same method can be found at multiple callsites, to avoid multiple compilations of the same method, most AOT compilers compile the methods in a particular order. Since the size and the content of the compilation units are not pre-compilation determined, the compilation order (i.e., *schedule*) itself is determined during the compilation. The usual compilation order starts from the entry points of the program, i.e., the methods in which the program execution starts, such as main or thread entry points. These methods are compiled first, then all their callees are transitively scheduled in a queue and compiled as previously described if they had not been compiled already. The algorithm in this article separates hot compilation units from the cold ones and, as such, has to modify the compilation process to ensure that no method would be compiled as cold if the heuristic determines that it is hot when invoked from another callsite. If the method were to be first compiled as cold, then that method could not be compiled again as a hot compilation unit. This is the reason why our algorithm first compiles all the hot compilation units and only then proceeds to compiling cold compilation units. This cannot be guaranteed only by taking the methods from the compilation queue following the original order.

The motivation for focusing on compilation-scheduling and inlining is that many optimizing compilers focus primarily on intraprocedural optimizations [23, 66, 95, 120]. For these compilers, inlining is an enabler for other compiler optimizations, because it extends "what the compiler sees." By having compilation units cover "hot code," we expect to benefit most from intraprocedural optimizations such as escape analysis [112], path-duplication [88], constant folding, and others [98].

Outline of the proposed solution. Given a set of partially context-sensitive profiles, some of which have higher execution counts than others (i.e., are "hotter") and thus represent small

fragments of the program regions that are globally hot, our algorithm is tasked with finding compilation units that cover "hot" parts of the program; in essence, to connect those hot fragments. While inlining algorithms typically start with an individual subroutine and extend its compilation unit in a *top-down* manner—by exploring the call tree from the compilation-unit root towards its callees until deciding to cease exploration and settling for a set of leaf nodes—our proposed algorithm works *bottom-up*: It first identifies the hottest partially context-sensitive entries, which represent hot compilation units. Our algorithm uses these profiles to form a forest of *breadcrumb trails*—trees that represent some portion of hot code. The algorithm then iteratively goes "upwards"—it extends the trails across likely callersand grafts together trails that share common methods.



The final set of breadcrumb trails represents hot entry points—compilation units for which additional compilation budget should be allocated. The breadcrumb trails are not used as actual inlining plans, but instead as inlining hints—they bias the inliner to explore and inline along those paths, because this leads to code in which most execution time is spent. Once these hot parts of the program are compiled, the rest of the program is compiled in a regular manner. Concretely, the inlining suggestions implied by the breadcrumb trails are mixed with the existing heuristics of the inliner.

3 COMPILATION-SCHEDULING AND INLINING ALGORITHM

In this section, we formally define the proposed compilation-scheduling and inlining algorithm. We present the algorithm in a top-down manner, starting with a high-level description, and we then gradually increase the level of detail.

Inputs. The input of the algorithm consists of the call graph of the program, along with the partial context-sensitive execution profiles. Each subroutine $s \in S$ of the program corresponds to one vertex in the *call graph* $G = (S, E \subseteq S \times S)$, and a call from subroutine s_1 to another subroutine s_2 corresponds to a directed edge $s_1 \rightarrow s_2$ of the graph. *Partial context-sensitive execution profile* is a mapping from calling contexts to execution counts, where a calling context represents some suffix of the program-counters on the call-stack, and the corresponding execution count represents how many times that call-stack occurred during the execution. Each *calling context* is a list of code locations $\ell_1, \ell_2, \ldots, \ell_n$ that are on the call-stack. Locations $\ell_1, \ell_2, \ldots, \ell_{n-1}$ represent the callsites at which the previous subroutine calls the next subroutine on the stack, and ℓ_n represents the specific code location in the callee at the top of the call-stack. The case when $n = 1$ is called a *context-insensitive profile*, and the case where, for every context, n corresponds to the number of calls on the call-stack, is called a *fully context-sensitive profile*.

Input simplifications. We defined the set of edges in the call graph as the subset of $S \times S$, where S is the set of subroutines in the program. This implies that each subroutine s_1 can call another subroutine s_2 on at most one callsite. In actual programs, a subroutine s_1 may contain multiple callsites to another subroutine s_2. We now show that the single-edge-per-vertex-pair limitation is inconsequential, because the multi-edged call graph can be reduced to the simpler representation.

Discussion. The implementation must accept a call graph in which each pair of vertices may be connected by any number of edges. The multi-edged graph can be transformed into a single-edged

graph in a way that the output of the algorithm in Listing 1 can be mapped back to a solution for the multi-edged graph. Each vertex V of the multi-edged call graph is translated to a subgraph with the V as the starting vertex of the subgraph. For each edge $V \rightarrow_i U$ at location i in the subroutine represented by the vertex V to another subroutine represented by a vertex U, a virtual vertex $V_{i,U}$ is inserted, and directed edges $V \rightarrow V_{i,U}$ and $V_{i,U} \rightarrow U$ are added. This transformation corresponds to outlining each callsite of the original program into a separate *bridge* method.

```
1   void foreach(int[] xs, int->void f) {
2     for (int i = 0; i < xs.length; i++)
3       f.apply(xs[i]);
4   }
5
6   void main(int[] args) {
7     min(args);
8     max(args);
9   }
10
11  int min(int[] xs) {
12    int m = Integer.MAX_VALUE;
13    foreach(xs, x -> if (x < m) m = x);
14    return m;
                          F.apply
15  }
16
17  int max(int[] xs) {
18    int m = Integer.MIN_VALUE;
19    foreach(xs, x -> if (x > m) m = x);
20    return m;
                          G.apply
21  }
```

Listing 1. Example Program.

The resulting compilation schedule that Algorithm 1 creates may mention virtual vertices $V_{i,U}$ in its edges $V \rightarrow V_{i,U}$ or $V_{i,U} \rightarrow U$. Since each vertex $V_{i,U}$ points to exactly one original vertex U, each $V \rightarrow V_{i,U}$ edge is replaced with $V \rightarrow_i U$. Similarly, since each virtual vertex $V_{i,U}$ is pointed to by exactly one vertex V, each $V_{i,U} \rightarrow U$ edge is replaced with $V \rightarrow_i U$. The transformed compilation schedule can have multiple edges at the same location i—these represent virtual calls.

In conclusion, any input call graph can be transformed into a call graph where every two vertices are connected with at most one edge. We will introduce a simplified formalization for clarity purposes, but it solves a problem that is equivalent to our concrete compilation problem.

Outputs. The output of the algorithm is a valid compilation schedule and a mapping from compilation units to the compilation budget. A *compilation schedule* is a directed graph, in which each vertex corresponds to a compilation unit, and each directed edge corresponds to a call from a compilation unit A to a compilation unit B. *Compilation unit* A is a subroutine S_A of the program, along with the subroutines inlined into A. A compilation schedule is valid if there exists a compilation unit whose root subroutine corresponds to the entry-point to the program and if there is a directed edge for each call (corresponding to the original program) between two compilation units.

Inlining [39, 63, 104] is an optimization in which the call to a subroutine is replaced with a copy of the code belonging to that subroutine. We say that a compilation unit A *calls* another compilation unit B if there is a code location in A that calls the root subroutine S_B of B. A compilation schedule is *valid* if there exists a compilation unit whose root subroutine corresponds to the entry-point to the programand if there is a directed edge for each call between two compilation units. *Compilation budget* is a function that assigns the amount of computational resources that an optimizing compiler is allowed to spend when compiling a particular compilation unit.

Example input. The program in Listing 1 computes the smallest and the largest integer from the given list of arguments. This program relies on a generic `foreach` subroutine that applies a lambda (i.e., function value) `f` to each integer in a given array. Starting with the `main` subroutine, the program consecutively calls the `min` and `max` subroutines, each of which invokes `foreach` with a lambda value that tracks the smallest or the largest integer, respectively. Note that, from the definition of the `foreach` subroutine, it is impossible to tell what is the exact implementation of the lambda `f`—this depends on where `foreach` is called from. We say that the call to `f` is *indirect* [91]. However, calls to `max`, `min` and `foreach` are *direct*.

The corresponding call graph is shown in Figure 1. We examine several profiles that track the target-subroutine invocation counts at the callsites. Each calling context is shown as a stack

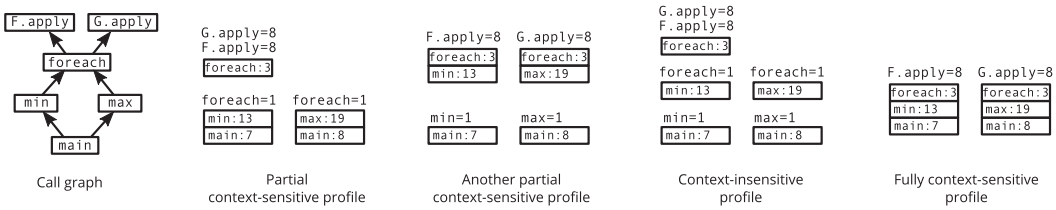


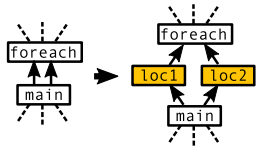
Fig. 1. Input examples for the algorithm.

of method-line-number pairs, and each calling context is mapped to the invocation counts of all the possible call targets. The two lambda values used by `min` and `max` are named `F` and `G`, respectively. The first partially context-sensitive profile "cuts" the call-stacks in the `min` and `max` subroutines, so the `foreach` calling context counts calls to both `F` and `G` lambdas. In the second partially context-sensitive profile, the calling contexts are "cut" in `main`, which makes the two `foreach` calling contexts more precise—it is now clear that `F` is only invoked when `foreach` gets called from `min`. We can conclude that the context-insensitive profile does not relay less information than the first partially context-sensitive profile in the following sense: Note that the call to `min` at `main:7` is direct, meaning that `main:7` is the only callsite of `min`—the entry `min:13` in the context-insensitive profile can be extended and merged with `main:7` (and similarly `max:19` with `main:8`), which yields a partially context-sensitive profile that represents *the same execution*. Furthermore, the fully context-sensitive profile does not relay more information than the second partially context-sensitive profile.

```
void foreach(int[] xs, int->void f) {
  for (int i = 0; i < xs.length; i++)
    f.apply(xs[i]);
}

void main(int[] args) {
  int min = Integer.MAX_VALUE;
  foreach(args, x -> if (x < m) m = x);
  int max = Integer.MIN_VALUE;
  foreach(args, x -> if (x > m) m = x);
}
```

Listing 2. Input Example.



Input simplification example. Listing 2 contains a modified version of the program from Listing 1. Instead of invoking methods `min` and `max` from the main method, their bodies are manually inlined directly into the `main` method. Thus, method `main` contains two callsites that target the same method `foreach`. In the adjacent figure, two virtual vertices `loc1` and `loc2` were added to the method `main` from Listing 2 as the callsites outlined into the bridge methods.

Example outputs. Figure 2 shows several possible outputs of the algorithm. Compilation schedule I consists of two compilation units—the compilation unit rooted at `main` inlines the subroutines `min` and `max`, each of which calls the second compilation unit rooted at `foreach`. Since the second compilation unit is called from two calling contexts, the `foreach` polymorphically inlines both implementations of `apply` [78]. In compilation schedule II, the `main` compilation unit does not inline any subroutines, so `min` and `max` are the roots of two separate compilation units, both of which inline a separate copy of `foreach`. The benefit of schedule II is that `foreach` can inline a single lambda and avoid a type-check. In compilation schedule III, no inlining is carried out—in this case, each compilation unit consists of a single subroutine. The last compilation schedule is not valid, because it does not include the edge for the call to `G.apply` (which can occur in execution).

High-level description. Algorithm 1 (described shortly) uses the call graph and the partially context-sensitive profile information to deduce which parts of the program are hot and to invest

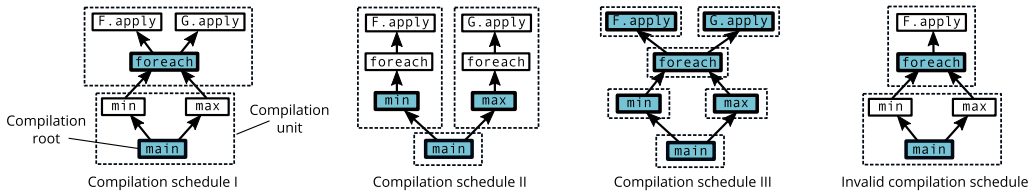


Fig. 2. Output examples for the algorithm.

ALGORITHM 1: Compilation Scheduling

input : call graph G, entry-point e, profile Π
output : compilation schedule Σ, budget B

$\Sigma = \varnothing$; $B = \varnothing$; $cold = \{ e \}$;
$hot = \textsc{DetectHot}(G, \Pi)$;
while $hot \neq \varnothing$ **do**
  $hot = hot \setminus \{ s_C \} : s_C \in hot$;
  $C = \textsc{InlineHot}(s_C)$;
  $\Sigma = \Sigma \cup \{ C \}$; $B = B \cup \{ C \}$;
  $cold = cold \setminus \{ s_C \}$;
  for $s \in callees(C) \setminus (\Sigma \cup hot)$ **do**
    if $\textsc{IsHot}(S)$ **then** $hot = hot \cup \{ s \}$;
    else $cold = cold \cup \{ s \}$;
  end
end
while $cold \neq \varnothing$ **do**
  $cold = cold \setminus \{ s_C \} : s_C \in cold$;
  $C = \textsc{InlineCold}(s_C)$;
  $\Sigma = \Sigma \cup \{ C \}$;
  for $s \in callees(C) \setminus (\Sigma \cup cold)$ **do**
    $cold = cold \cup \{ s \}$;
  end
end

more effort into compiling those parts. To do this, it separates compilation units into *hot* and *cold*. In this article, the mapping from compilation units to the compilation budget is binary—if a compilation unit is hot, then it gets an increased compilation budget, and if it is cold, then it gets the default budget. The motivation is to increase the degree of optimization in code that is frequently executed, but avoid bloating the size of the code that does not significantly contribute to the total execution time.

The algorithm maintains two queues—*hot* and *cold*. It first picks the initial set of hot subroutines and places them on the hot queue. Then, it iteratively removes a hot subroutine and performs inlining to create the compilation unit that starts with that hot subroutine. If, after inlining, the hot compilation unit calls other subroutines (i.e., callees) that were not previously pushed to the queue, then the algorithm places each such callee on either the hot queue or the cold queue. After the hot queue becomes empty, the algorithm schedules the remaining cold compilation units (including the program entry-point) and iteratively removes the cold subroutines until the cold queue is empty. This high-level description is captured in Algorithm 1, while more detailed information is provided in the following subsections.

Example execution. To see how the algorithm works, consider the program from the earlier example, whose compilation is in Figure 3. The algorithm first puts the `foreach` subroutine to the hot list and the `main` subroutine (i.e., program entry-point) to the cold list. It then removes `foreach` from the hot list, inlines `F.apply` and decides that there is no budget left for inlining `G.apply`. The algorithm nevertheless concludes that `G.apply` is hot and places it on the hot list. After the hot list becomes empty, the algorithm schedules `main` and refrains from inlining due to low frequency of the callsites—`min` and `max` are pushed to the cold list. Importantly, the algorithm does not inline hot compilation units into cold compilation units—neither `min` nor `max` inlines `foreach`.

Discussion. There are several important details that we must clarify. First, the algorithm needs a concrete set of steps to decide on the initial set of hot subroutines. Second, subroutines must be expanded to compilation units with a concrete inlining policy. Third, after a hot compilation unit is formed, the algorithm must decide which of the remaining callees are hot and which are cold.

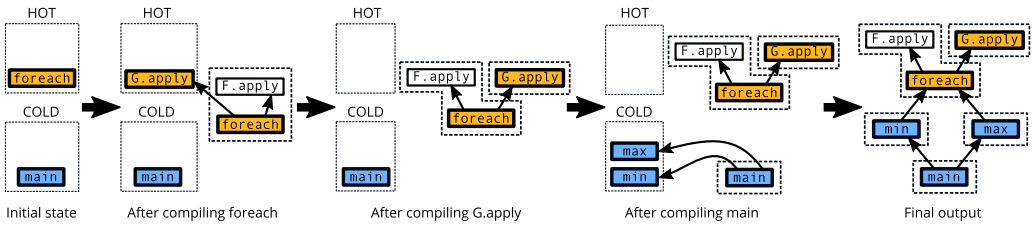


Fig. 3. Example of the compilation-ordering.

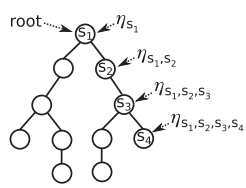
In Algorithm 1, these concerns are captured with procedures DetectHot, InlineHot, InlineCold, and IsHot, respectively. To drive the behavior of these procedures, our algorithm estimates which subroutines are hot in a particular calling context. A subroutine is *hot* either if the estimated time spent exclusively in that subroutine (without the callees) exceeds some fixed percentage ψ of the total time spent in the program or if the subroutine transitively calls other hot subroutines up to some point determined by the compilation budget. The rest of this section contains a detailed description of how the aforementioned procedures work. The specifics of the algorithm are thus divided into three components:

(1) DetectHot: the procedure that constructs breadcrumb trails using the call graph G and the partial context-sensitive profile P, described in Section 3.3.
(2) InlineHot and InlineCold: the modified inlining algorithm guides its inlining decisions using the breadcrumb-trail information, described in Section 3.5.
(3) IsHot: the procedure that uses the breadcrumb trails to separate callees of a compilation unit into hot and cold subroutines, described in Section 3.6.

3.1 Call Trees

To precisely define breadcrumb trails, we first express what we presume under the term call tree.

Notation. Before we define call trees, we adopt the following scheme for referring to nodes within trees. Let us assume that every node in the tree is associated with some subroutine s (multiple nodes may be associated with the same subroutine). The notation $\eta_{s_1,s_2,\ldots,s_n}$ then refers to a node that is reached by starting from the root of the tree (whose subroutine is s_1) and following the sequence of nodes whose subroutines are s_2, s_3, and so on, until reaching a node whose subroutine is s_n. In other words, we treat the tree as a prefix tree [73] and the sequence $s_1, s_2, \ldots, s_n$ as the prefix stored in the tree. For our purposes, the node $\eta_{s_1,s_2,\ldots,s_n}$ will represent a call stack of subroutines. This is illustrated in the adjacent figure.



The notation η or η_x, which does not have a sequence in the subscript, refers to any node in the tree and is used when the prefix of the node is not important in the discussion. When the subscript is a single-element sequence, we take care to disambiguate what we mean in the text.

Call graphs. For a specific call graph G, we rely on a function $callees_G(s)$ that returns the set of subroutines that s can invoke, according to the call graph G. Note that, in the previous example, $callees_{Listing\ 1}$(`foreach`) consists of `F.apply` and `G.apply`.

$$callees_G(s) \equiv \{s_2 : s \rightarrow s_2 \in E\} \qquad where \quad G = (S, E \subseteq S \times S) \tag{1}$$

For readability, we omit the index G later in the text—we always mean "the G that is compiled."

We define the root of a call tree as the ancestor node of all the other nodes (*common infimum*):

$$root(N, E) = \inf_{E^*} N. \tag{2}$$

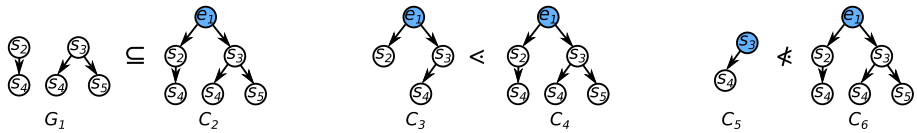
We now introduce two partial orders on (infinite) graphs. The *subset* relation between graphs $C_1 = (N_1, E_1)$ and $C_2 = (N_2, E_2)$ is defined as follows:

$$(N_1, E_1) \subseteq (N_2, E_2) \equiv N_1 \subseteq N_2 \wedge E_1 \subseteq E_2. \tag{3}$$

Graph $C_1 = (N_1, E_1)$ *nests within* the graph $C_2 = (N_2, E_2)$ if and only if E_1 and E_2 induce partial orders on N_1 and N_2 (i.e., C_1 and C_2 are directed acyclic graphs), C_1 and C_2 share a common infimum (i.e., they have a *common root*), and C_1 is a subset of C_2 (note that notation $root(N, E)$ implies that the *unique* least element of N exists, further implying that C_1 and C_2 must be connected and acyclic):

$$(N_1, E_1) \lessdot (N_2, E_2) \equiv root(N_1, E_1) = root(N_2, E_2) \wedge (N_1, E_1) \subseteq (N_2, E_2). \tag{4}$$

The subset and nests-within relations are illustrated by the examples in the following figure. The graph G_1 (which consists of two disconnected components) is not a call tree, but is a subset of the call tree C_2. The call tree C_3 is nested within C_4, but the call tree C_5 is not nested within C_6, because they do not share a common root.



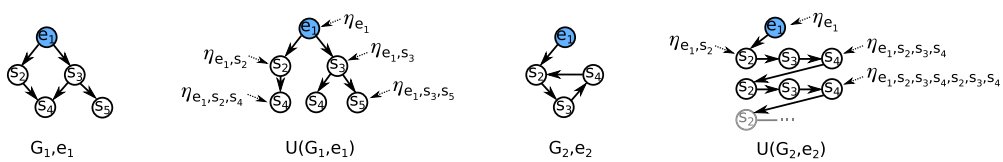
Call trees. Next, we introduce the term *call tree*. Informally, a call tree is a prefix tree of all the possible call stacks of a given program.

Let a program $P = (G, e)$ consist of the call graph G and the entry-point subroutine e. The *unfolding of a program P*, $U(P)$ is a tuple (N, E) composed of a set of nodes N and a set of edges E, in which the root node corresponds to the entry point e of P, each node $\eta_{e,\ldots,s_n}$ corresponds to a call stack ending with the subroutine $s_n = sub(\eta_{e,\ldots,s_n})$, and the set of children of $\eta_{e,\ldots,s_n}$ is $\eta_{e,\ldots,s_n,s_{n+1}}$ such that $s_{n+1} \in callees_G(s_n)$.

$$\begin{aligned} U(P) \equiv (N, E \subseteq N \times N) \quad & where \quad \eta_e \in N \wedge \\ & (s_{n+1} \in callees_G(s_n) \Leftrightarrow (\eta_{e,\ldots,s_n} \in N \Leftrightarrow \eta_{e,\ldots,s_n,s_{n+1}} \in N)) \wedge \\ & (s_{n+1} \in callees_G(s_n) \Leftrightarrow (\eta_{e,\ldots,s_n} \in N \Leftrightarrow \eta_{e,\ldots,s_n} \rightarrow \eta_{e,\ldots,s_n,s_{n+1}} \in E)) \end{aligned} \tag{5}$$

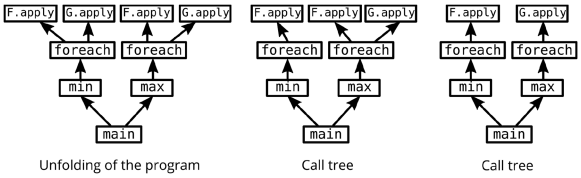
Note that any call tree C is nested within the unfolding $U(P)$ of the program, that is, $C \lessdot U(P)$.

The following figure shows two call graphs, and their corresponding call-tree examples, For a call graph on the left, the corresponding call tree is finite, while in case of the call graph on the right, because it contains a cycle, a call tree is potentially infinite, which we demonstrate on the rightmost tree in the figure.



Example. The previous figure illustrates call trees and the unfolding of a program on several examples. These examples correspond to the program from Listing 1. The leftmost tree represents the unfolding of a program, and the remaining two trees represent the examples of the call trees, which nest within the first tree.

3.2 Breadcrumb Trails

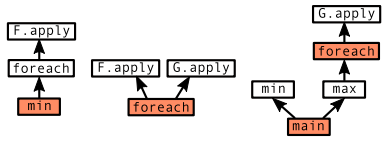


Now, we define the *breadcrumb trails*. Informally, they are similar to call trees, except they do not need to start with the entry point of the program. In other words, they do not need to be nested in a call tree per Equation (4). Instead, a *breadcrumb trail* of the program P is any finite, connected subgraph of some call tree of the program P. We define the set of breadcrumb trails of program P as follows (below, $\aleph_0$ is the size of the set of natural numbers $\mathbb{N}$, so we use $|N| < \aleph_0$ to say "finite"):

$$\mathbb{B}(P) \equiv \{(N, E) : |N| < \aleph_0 \wedge E \subseteq N \times N \wedge \exists x, x = root(N, E) \wedge \exists C \in \mathbb{C}(P), (N, E) \subseteq C\}. \quad (6)$$

By Equation (6), a breadcrumb trail β is generally not a call tree, and its main restriction is that the children of each node η are from $callees(sub(\eta))$. The breadcrumb trail $\beta = (N, E)$ is, however, a tree, with a well-defined root $root(N, E)$. A node in the breadcrumb trail is called a *breadcrumb*. Note, that a breadcrumb trail is not just a call tree because each node in the call tree must contain all the callees of the corresponding subroutine, whereas in a trail it does not.

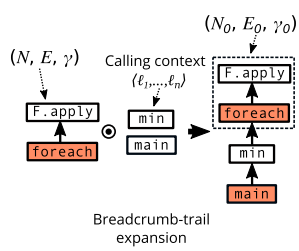
Consider the example breadcrumb trails shown below. The trail with the `min` root leads up to `F.apply`, while the trail with the `foreach` root leads to both `F.apply` and `G.apply`—both of them serve as recipes for finding the hot subroutines starting from specific calls. In this article, the use-case for breadcrumb trails is, loosely speaking, to transitively connect a call to its "hot" callees, that is, subroutines in which most execution time is spent. However, the definition does not mandate that the subroutines are "hot"—a breadcrumb trail could be used for other purposes, too.



Note that, in the figure, some nodes are annotated with the orange color. The root is always annotated with orange, but any other node of the breadcrumb trail can also be annotated. These annotated nodes are called *graft points*, and the respective trails are called *annotated breadcrumb trails*. The set of annotated breadcrumb trails of a program P are defined as follows:

$$\mathbb{T}(P) \equiv \{(N, E, \gamma) : (N, E) \in \mathbb{B}(P) \wedge \gamma \subseteq N \wedge root(N, E) \in \gamma\}. \quad (7)$$

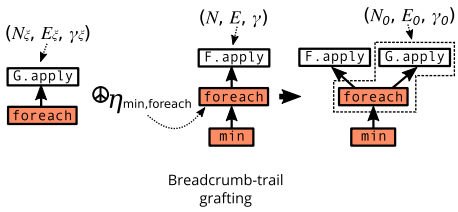
In the remaining text, we will use the term *trail* to refer to annotated breadcrumb trails. The purpose of the graft points will be to annotate the places where two trails can be grafted to each other.



Operations. We now define operations on the annotated breadcrumb trails, which are central in the rest of the algorithm. Assume that we have a relatively short breadcrumb trail—it only describes the path to the hot code from callers that are "very close." Given such a trail $\tau = (N, E, \gamma)$ that starts with $root(\tau) = root(N, E)$, it is of interest to expand it with a calling context $\ell_1, \ldots, \ell_n$. The operation $\odot$, called *breadcrumb-trail expansion*, produces a new trail that starts with a chain of nodes that correspond to $\ell_1, \ldots, \ell_n$ and in which node ℓ_n points to $root(\tau)$. The restriction here is that ℓ_n calls $sub(root(\tau))$. In the adjacent figure, the trail rooted at `foreach`

is expanded to a trail rooted at `main`, allowing the hot code to be detected from "further away."

$$
\begin{aligned}
(N, E, \gamma) \odot \langle \ell_1, \ldots, \ell_n \rangle \equiv (&\{\eta_{\ell_1, \ldots, \eta_{\ell_1, \ldots, \ell_n}}\} \cup N_0, \{\eta_{\ell_1} \rightarrow \eta_{\ell_1, \ell_2, \ldots, \eta_{\ell_1, \ldots, \ell_n}} \rightarrow root(N_0, E_0)\} \cup E_0, \eta_{\ell_1} \cup \gamma_0) \\
N_0 = \{\eta_{\ell_1, \ldots, \ell_n, s_1, \ldots, s_n} &: \eta_{s_1, \ldots, s_n} \in N\} \quad \gamma_0 = \{\eta_{\ell_1, \ldots, \ell_n, s_1, \ldots, s_n} : \eta_{s_1, \ldots, s_n} \in \gamma\} \\
E_0 = \{\eta_{\ell_1, \ldots, \ell_n, s_1, \ldots, s_n} &\rightarrow \eta_{\ell_1, \ldots, \ell_n, s_1, \ldots, s_n, s_{n+1}} : \eta_{s_1, \ldots, s_n} \rightarrow \eta_{s_1, \ldots, s_n, s_{n+1}} \in E\}
\end{aligned}
\tag{8}
$$



In Equation (8), existing nodes are renamed so they include the $\ell_1, \ldots, \ell_n$ prefix, and a new set of nodes and directed edges is additionally included for each call in the calling context $\ell_1, \ldots, \ell_n$.

Assume now that we have several trails that contain a breadcrumb for the same subroutine s. When considering which callees in s are "important" (as explained later), one needs to inspect the breadcrumbs for s in all such trails. This is inconvenient—it is much easier for a compiler to inspect a single breadcrumb whenever it needs to decide is "important." Therefore, we define another breadcrumb-trail operation $\circledast_\eta$, called *grafting*, which attaches a trail $(N_\xi, E_\xi, \gamma_\xi)$ to a node η within another trail (N, E, γ). In the adjacent figure, grafting adds the callsite of `G.apply` to the `foreach` node within the `min` trail. The grafting in this particular example is correct, albeit not too useful in practice—in our program, method `G.apply` is obviously never invoked from the calling context that starts with `min`.

More formally, given a target trail $\tau = (N, E, \gamma)$, its breadcrumb $\eta_{s_1, \ldots, s_n} \in \gamma$ and a trail $\xi = (N_\xi, E_\xi, \gamma_\xi)$ such that $sub(\eta_{s_1, \ldots, s_n}) = sub(root(\xi))$, i.e., $s_n = sub(root(N_\xi, E_\xi))$, the grafting $\xi \circledast_{\eta_{s_1, \ldots, s_n}} \tau$ produces a copy of τ in which the node $\eta_{s_1, \ldots, s_n}$ has an additional subtree from ξ for each child node $\eta_{s_n, c_2} \in children(root(\xi))$ such that $\nexists \eta_{s_1, \ldots, s_n, c_2} \in children(\eta_{s_1, \ldots, s_n})$, while all the subtrees of the children $\eta_c \in children(root(\xi))$ for which $\exists \eta_{s_1, \ldots, s_n, c_2} \in children(\eta_{s_1, \ldots, s_n})$ are recursively grafted.

$$
\begin{aligned}
(N_\xi, E_\xi, \gamma_\xi) \circledast_{\eta_{s_1, \ldots, s_n}} (N, E, \gamma) \equiv (N \cup N_0, E \cup E_0, \gamma \cup \gamma_0) \quad &where \quad root(N_\xi, E_\xi) = s_n \quad \eta_{s_1, \ldots, s_n} \in \gamma \\
N_0 = \{\eta_{s_1, \ldots, s_n, c_2, \ldots, c_m} &: \eta_{s_n, c_2, \ldots, c_m} \in N_\xi\} \quad \gamma_0 = \{\eta_{s_1, \ldots, s_n, c_2, \ldots, c_m} : \eta_{s_n, c_2, \ldots, c_m} \in \gamma_\xi\} \\
E_0 = \{\eta_{s_1, \ldots, s_n, c_2, \ldots, c_m} &\rightarrow \eta_{s_1, \ldots, s_n, c_2, \ldots, c_m, c_{m+1}} : \eta_{s_n, c_2, \ldots, c_m} \rightarrow \eta_{s_n, c_2, \ldots, c_m, c_{m+1}} \in E_\xi\}
\end{aligned}
\tag{9}
$$

If we adopt the convention that $(N_\xi, E_\xi, \gamma_\xi) \cup (N, E, \gamma) \equiv (N_\xi \cup N, E_\xi \cup E, \gamma_\xi \cup \gamma)$ and that $root(N_\xi, E_\xi, \gamma_\xi) \equiv root(N_\xi, E_\xi)$, then we can define grafting more concisely as follows:

$$
\xi \circledast_{\eta_{s_1, \ldots, s_n}} (N, E, \gamma) \equiv (\xi \odot \langle s_1, \ldots, s_{n-1} \rangle) \cup (N, E, \gamma) \quad where \quad root(\xi) = s_n \quad \eta_{s_1, \ldots, s_n} \in \gamma. \tag{10}
$$

Table 1 summarizes the most important symbols introduced so far. Having defined the basic trail operations, we can now describe the hot-code detection algorithm.

3.3 Hot-code Detection

We established that the code from Algorithm 1 separates the call-graph into hot and cold regions. The hot regions of the call graph consist of code that is frequently executed, along with the calling context that is sufficiently large to include the relationships required for code optimizations. The purpose of the breadcrumb trails from the previous section is to delimit these hot regions.

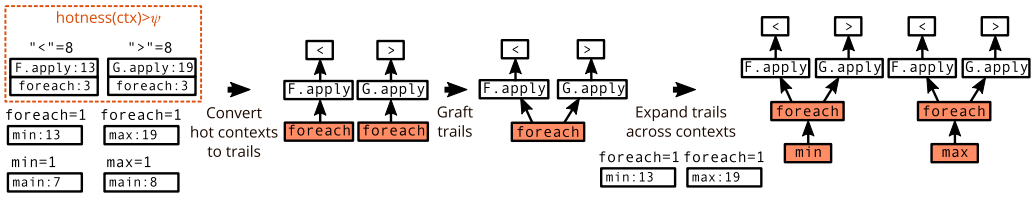
The breadcrumb trails that delimit the hot code, which the scheduling algorithm relies on, must be derived from the program's execution profile. Consider the task of constructing the trails from a fully context-sensitive profile—one way to achieve this would be to create trails for the set of calling contexts whose execution count is above a threshold ψ and to then graft them all together.

Table 1. Summary of Key Symbols from Sections 3.1 and 3.2

Symbol	Name	Meaning
(G, e)	Program	Call graph G with a distinguished entry-point e.
$callees_G(s)$	Callees (Equation (1))	Set of subroutines that can be invoked from a subroutine s in a call-graph G (i.e., outgoing edges of s).
$\eta_{s_1,\ldots,s_n}$	Node (in a tree structure)	A node that corresponds to a call stack of subroutines s_1, s_2, $\ldots$, s_n in the call-tree (or a trail) rooted at s_1.
$C_1 \lessdot C_2$	Nests within (Equation (4))	Tree C_1 nests within the tree C_2 if they have the common root, and tree C_1 is a subset of C_2.
$U(G, e)$	Unfolding of a program (G, e) (Equation (5))	Tree in which the root is the entry point e, and each node $\eta_{e,\ldots,s_n}$ has a set of children $\eta_{e,\ldots,s_n,s_{n+1}}$, where $s_{n+1} \in callees_G(s_n)$.
$C(G, e)$	Call tree	Prefix tree of all the possible call stacks of a given program (G, e).
$\mathbb{B}(G, e)$	Breadcrumb trails (Equation (6))	Set of finite trees $\beta = (N, E)$ such that β is a subset of some call tree C, i.e., $\beta \subseteq C(G, e)$.
$\mathbb{T}(G, e)$	Annotated trails (Equation (7))	Set of trees $\tau = (N, E, \gamma)$, such that each trail $\beta = (N, E)$ is annotated with a set of graft points $\gamma \subseteq N$.
$\tau \odot \langle \ell_1, \ldots, \ell_n \rangle$	Trail expansion (Equation (8))	Operation that attaches the calling context $\ell_1, \ldots, \ell_n$ on top of the root of the trail τ.
$\xi \circledast_{\eta_{s_1,\ldots,s_n}} \tau$	Trail grafting (Equation (9))	Operation that attaches the trail ξ to an existing node $\eta_{s_1,\ldots,s_n}$ within another trail τ.

However, the input to our algorithm is a *partially* context-sensitive profile. The hottest calling contexts in such a profile are typically too short to connect code regions in a useful manner, so the corresponding trails must be expanded until the covered part of the call tree is sufficiently large for subsequent compiler optimizations to potentially improve performance. The algorithm is thus posed with several challenges: selecting a good set of initial calling contexts, choosing the beneficial calling contexts to speculatively expand across, and knowing when to stop the expansion $\odot$ and grafting $\circledast$ operations. We call this part of the algorithm *hot-code detection.*

High-level description. The algorithm starts by selecting a set of initial hot contexts and converts them to trails. These trails are grafted together wherever this is possible. The following process is then repeated until reaching a termination condition (Section 3.4): The most favorable trail is selected and expanded across calling contexts in which it is hot. These expanded trails are then grafted into other trails from the trail set or placed back into the trail set if grafting is not possible.



Example detection. Consider the scenario in the preceding figure. The initial set of profiles (on the left) is partially context-sensitive, so some profiles have the context length $n = 1$, and some have $n = 2$. The algorithm first picks calling contexts whose profile-counts exceed a predetermined value ψ: these are `foreach`→`F.apply`→'<' and `foreach`→`G.apply`→'>'. These calling contexts are then converted to the initial set of breadcrumb trails. In the next step, each trail ξ is grafted onto nodes $\eta_{s_1,\ldots,s_n}$ of other trails such that $sub(root(\xi)) = sub(\eta_{s_1,\ldots,s_n})$—in the preceding example, the

first `foreach` trail (which calls `F.apply`) is grafted to the second `foreach` (which calls `G.apply`). The algorithm then searches the profiles to identify the possible callers of `foreach`, determines that `min` and `max` are the most common callers, and expands `foreach` along each of these calling contexts. The two aforementioned steps, grafting and expansion, are repeated consecutively until the algorithm decides that it ran out of budget. In this example, the algorithm stops after producing two trails that are rooted at the `min` and `max` subroutines.

Note that the calling contexts in the previous example are sufficiently long to allow static analysis to determine that `G.apply` is never called from `min` and that `F.apply` is never called from `max`. In other words, the resulting trails are not always nested within a minimum call tree. We do not remove unreachable calls during hot-code detection, because compilers have optimizations that remove such calls after callees get inlined into a single compilation unit [43, 84, 87, 112, 118]; moreover, state-of-the-art inlining algorithms use dataflow analyses to simplify the call tree before inlining happens [62, 81, 97, 107]. In other words, we later ensure that the inliner neither inlines `G.apply` from `min`'s calling context, nor `F.apply` from `max`'s calling context, even though the trails contain these calls. Therefore, we do not simplify the trails during hot-code detection, since in our implementation, the inliner prunes the trail later.

Trail-set operations. To formally describe the algorithm, we define two convenience operations on sets of trails, called union-grafting $\overset{\circledast}{\cup}$ and self-root-grafting $\overset{\circledast}{\nabla}_{root}$. *Union-grafting* operation merges two sets of trails together, while the *self-root-grafting* operation grafts a single trail from the set onto other trails from the same set.

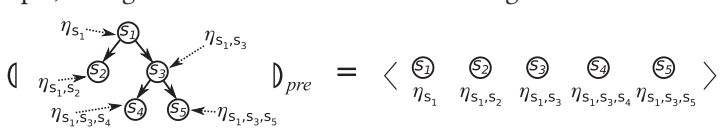
To determine the order in which certain operations are performed, we will choose a specific ordering for the nodes within trails and the trails within trail sets. Let $(\!|X|\!)_{ord}$ be a sequence of elements from the set X, which are ordered according to the total order *ord*:

$$(\!|X|\!)_{ord} \equiv \langle x_1, \ldots, x_n \rangle : x_i \in X \wedge x_i <_{ord} x_j \Rightarrow i < j \wedge |X| = n. \tag{11}$$

Next, let $pre(\tau)$ be the total order of the nodes in trail τ, induced by the left-to-right preorder traversal of τ. The preorder is the lexicographic ordering of the call stacks represented by the nodes:

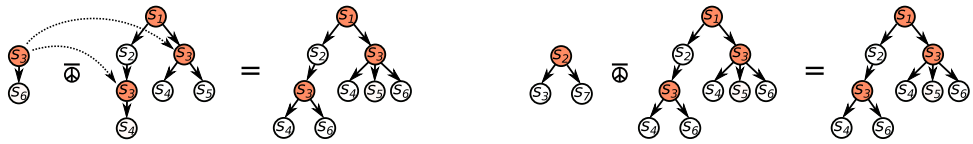
$$\eta_{s_{1,1},\ldots,s_{1,n_1}} <_{pre(\tau)} \eta_{s_{2,1},\ldots,s_{2,n_2}} \equiv s_{1,1} \ldots s_{1,n_1} <_{\text{lex}} s_{2,1} \ldots s_{2,n_2}. \tag{12}$$

As an example, the figure below shows the node ordering of a trail with five nodes:



Union-grafting $\overset{\circledast}{\cup}$ relies on the right-associative helper operation $\overline{\circledast}$, which is similar to the grafting operation $\circledast_\eta$, the difference being that it grafts to all possible graft points instead of a specific point η. In other words, given a graftee trail $\xi = (N_\xi, E_\xi, \gamma_\xi)$ and a target trail $\tau = (N, E, \gamma)$, $\xi \overline{\circledast} \tau$ grafts ξ to *all* candidate graft-points in τ, i.e., to all graft-points that represent the same subroutine as the root of ξ. The graft-points are ordered in preorder traversal of the trail τ.

The following figure shows two examples of the $\overline{\circledast}$ operation. On the left, the trail $s_3 \rightarrow s_6$ is grafted to both graft points s_3 of the target trail. On the right, the trail $s_3 \leftarrow s_2 \rightarrow s_7$ cannot be grafted to any graft points in the target trail, because the target trail has no graft point with s_2.



The definition of the $\overline{\circledast}$ operation uses a helper function g to recursively graft at all candidates:

$$\xi \overline{\circledast} (N, E, \gamma) \equiv g(\xi, (N, E, \gamma), \gamma)$$

$$g(\xi, (N, E, \gamma), B) \equiv \begin{cases} \xi \circledast_{\eta_{last}} g(\xi, (N, E, \gamma), B \setminus \eta_{last}) & \text{if } B \neq \varnothing \wedge s_{m,n_m} = sub(root(N_\xi, E_\xi)) \\ g(\xi, (N, E, \gamma), B \setminus \eta_{last}) & \text{if } B \neq \varnothing \wedge s_{m,n_m} \neq sub(root(N_\xi, E_\xi)) \\ (N, E, \gamma) & \text{otherwise} \end{cases}$$

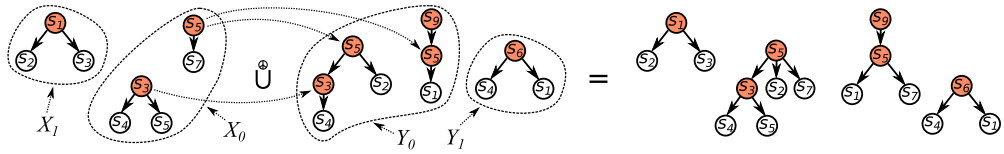
$$\text{where} \quad (\!|B|\!)_{pre(N,E,\gamma)} = \langle \eta_{s_{1,1},\ldots,s_{1,n_1}}, \ldots, \eta_{s_{m,1},\ldots,s_{m,n_m}} \rangle \qquad \eta_{last} = \eta_{s_{m,1},\ldots,s_{m,n_m}}. \tag{13}$$

Finally, to establish an ordering on sets of trails, we define the ordering *lex* on *trails* τ_1 and τ_2 as the lexicographic ordering of the lists obtained with a preorder traversal of the nodes of τ_1 and τ_2:

$$(N_1, E_1, \gamma_1) <_{lex} (N_2, E_2, \gamma_2) \equiv (\!|N_1|\!)_{pre(N_1,E_1,\gamma_1)} <_{lex} (\!|N_2|\!)_{pre(N_2,E_2,\gamma_2)}. \tag{14}$$

In Equations (12) and (14), we defined lexicographic orderings to order the nodes and the trails, as this allows defining subsequent operations deterministically. Without any loss of generality, we could have picked different orderings, but in the formalization, we adhere to *lex* for simplicity.

Union-graft operation $\overset{\circledast}{\cup}$. Assume that we want to create a union of two sets of trails X and Y but to avoid having multiple trails with the same root subroutine in the result. The union-graft operation achieves this by grafting trails from X to trails in Y wherever possible and then creating a union with the remainder. An example of $X \overset{\circledast}{\cup} Y$ is shown in the following figure:



To union-graft a trail set X into a trail set Y, the set X can be separated into two sets X_0 and X_1, so trails from X_0 can be grafted to trails in $Y_0 \subseteq Y$, and trails from X_1 cannot be grafted to trails in Y. The union-graft operation grafts each trail $\xi_i \in X_0$ to the graft sites within trails $\tau_j \in Y_0$ in the lexicographic order of the trails ξ_i. The result of the grafting is unified with the remainders $Y_1 = Y \setminus Y_0$ and X_1 (below, $\cup$ is disjoint union):

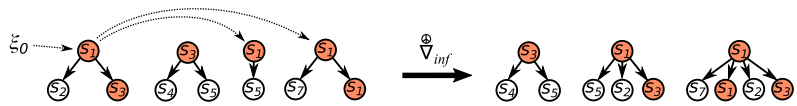
$$X \overset{\circledast}{\cup} Y \equiv \begin{cases} X_1 \cup Y_1 \cup \left(\{\xi_1 \overline{\circledast} \ldots \overline{\circledast} \xi_n \overline{\circledast} \tau_1\} \cup \left(X_0 \overset{\circledast}{\cup} (Y_0 \setminus \{\tau_1\})\right)\right) & \text{if} \quad Y_0 \neq \varnothing \\ X_1 \cup Y_1 & \text{if} \quad Y_0 = \varnothing \end{cases}$$
$$\text{where} \quad X = X_0 \cup X_1 \quad Y = Y_0 \cup Y_1 \quad (\!|X_0|\!)_{lex} = \langle \xi_1, \ldots, \xi_n \rangle \quad (\!|Y_0|\!)_{lex} = \langle \tau_1, \ldots, \tau_m \rangle$$
$$X_0 = \{\xi \in X : \exists \tau \in Y, \xi \overline{\circledast} \tau \neq \tau\} \quad Y_0 = \{\tau \in Y : \exists \xi \in X, \xi \overline{\circledast} \tau \neq \tau\}. \tag{15}$$

The preceding definition says the following: For all the trails $\xi \in X$ that can be grafted to at least one trail $\tau \in Y$, graft them to all trails $\tau \in Y$ to which they can be grafted. Leave the remaining trails $\xi \in X$ unchanged and produce a union of all these trails.

Self-root-graft operation $\overset{\circledast}{\nabla}_{root}$. Self-root-graft operation takes a single trail set T as an input and grafts one of its trails τ_0 to the roots of other trails from T. In Equation (16), it is defined as following: If there is a candidate trail τ_0 in T that can be grafted to a root of another trail, then it is grafted to the roots of any such trail from T. Otherwise, the result is the original trail set T. The self-root-graft operation is illustrated in the following example, in which trail $s_2 \leftarrow s_1 \rightarrow s_3$ is

grafted onto the roots of two other trails, but not onto any other graft points with the subroutine s_1:

$$\overset{\circledast}{\nabla}_{root} T \equiv \begin{cases} \{\xi_0 \underset{root(\tau)}{\circledast} \tau : \tau \in R(\xi_0, T)\} \cup (T \setminus R(\xi_0, T)) & \langle \xi_0, \ldots, \xi_n \rangle = (\!|\{\xi \in T : R(\xi, T) \neq \varnothing\}|\!)_{lex} \\ T & otherwise \end{cases}$$
$$where \quad R((N_\xi, E_\xi, \gamma_\xi), T) = \{(N, E, \gamma) \in T : \tau \neq (N_\xi, E_\xi, \gamma_\xi) \wedge root(N, E) = root(N_\xi, E_\xi)\}. \tag{16}$$



The fix-point (fix $\overset{\circledast}{\nabla}_{root})(T)$ represents the limit of the repetitive application of the self-root-grafting operation for the trail set T. Below, notation $\circ^n$ represents n applications of a function ($\lim_{n\to\infty}$ is a ε-definition of the limit on a metric $d(X, Y)$ that is 0 if and only if $X = Y$ and is a positive constant otherwise):

$$(\text{fix } \overset{\circledast}{\nabla}_{root})(T) = \lim_{n\to\infty} (\circ^n \overset{\circledast}{\nabla}_{root})(T). \tag{17}$$

Consider the previous example with the self-root-graft operation. If we apply it one more time, then the trail $s_1 \to \{s_5, s_2, s_3\}$ will be grafted onto the root of the remaining trail with the same root subroutine s_1 ($s_1 \to \{s_7, s_1, s_2, s_3\}$), but subsequent applications of $\overset{\circledast}{\nabla}_{root}$ have no effect. In that example, the limit exists and $n = 2$. We now show that the limit always exists.

Lemma 3.1. *The fixpoint (fix $\overset{\circledast}{\nabla}_{root})(T)$ exists for all T and is reached in a finite number of steps.*

Proof. Consider the grafting operation $\circledast_{\eta_{s_1,\ldots,s_n}}$: By Equation (9), given two input trails, the result of the operation is a single trail. By examining the first case in Equation (16), we note that the resulting set is always smaller for this case, because we picked the trail ξ_0 in a way that there is a set od trails τ, which share the same root subroutine as ξ_0, which allows the grafting. Thus, the cardinalities of the trail sets in the sequence $(\circ^i \overset{\circledast}{\nabla}_{root})(T)$ are strictly monotonically decreasing. Consequently, the second case must be eventually applied (because the trail-set size cannot be less than 1), so there is an n for which $(\circ^{n+1} \overset{\circledast}{\nabla}_{root})(T) = (\circ^n \overset{\circledast}{\nabla}_{root})(T)$, which implies that the limit exists and that the fixpoint is equal to $(\circ^n \overset{\circledast}{\nabla}_{root})(T)$. □

Lemma 3.2. *Each trail (N, E, γ) in (fix $\overset{\circledast}{\nabla}_{root})(T)$ has a unique root subroutine.*

Proof. By Equation (16), the fixpoint cannot be reached if there are remaining candidates ξ_0 for grafting. As long as two trails in $(\circ^i \overset{\circledast}{\nabla}_{root})(T)$ share the root subroutine, the first case applies, and there is a candidate ξ_0 such that $R(\xi_0, (\circ^i \overset{\circledast}{\nabla}_{root})(T)) \neq \varnothing$, so $(\circ^{i+1} \overset{\circledast}{\nabla}_{root})(T) \neq (\circ^i \overset{\circledast}{\nabla}_{root})(T)$. Table 2 contains the most important symbols presented in Section 3.3. □

Algorithm. Having defined the necessary machinery, we can now formally and concisely describe the hot-code detection algorithm that was outlined in the earlier example. The DetectHot procedure, shown in Algorithm 2, first picks a set of hot contexts from the profile Π and converts them to a trail set T_0 in line 1. The concrete steps for this are abstracted in the procedure InitialTrails. Since the trails in T_0 correspond directly to profiles, it is usually possible to graft some of them together, as seen in the previous example. This initial grafting repetitively picks some trail from T_0 and grafts it onto the remaining trails (the self-root-grafting operation $\overset{\circledast}{\nabla}_{root}$). The initial-grafting is represented with the operation fix $\overset{\circledast}{\nabla}_{root}$ in line 2, and it produces the trail set T. The algorithm then creates an empty trail set F and proceeds as follows: As long as a termination condition DetectionDone is not met, the algorithm selects a trail τ according to some policy TopTrail in line 5 and calls a procedure CallingContexts to identify the set Γ of calling contexts that call

Table 2. Summary of Key Symbols from Section 3.3

Symbol	Name	Meaning
$(\!\lvert X \rvert\!)_{ord}$	Ordering of the set X (Equation (11))	Sequence of elements from set X, ordered by the total order ord.
$\xi \overline{\circledast} \tau$	Graft-to-all operation (Equation (13))	Operation that grafts the trail ξ to all candidate graft points in τ.
$X \overset{\circledast}{\cup} Y$	Union-graft operation (Equation (15))	Grafts a subset $X_0 \subseteq X$ of trails from X to Y wherever possible and then returns a union with the remaining trails $X \setminus X_0$.
$\overset{\circledast}{\nabla}_{root} T$	Self-root-graft operation (Equation (16))	Picks a trail τ_0 from the trail set T and grafts it to other trails from T that have the same root subroutine.
$(\text{fix } \overset{\circledast}{\nabla}_{root})(T)$	Fixpoint of $\overset{\circledast}{\nabla}_{root}$ (Equation (17))	The limit of the repetitive application of the self-root-graft.

$sub(root(\tau))$ in line 6. The trail τ is expanded along each of the calling contexts $c \in \Gamma$ to a new context $\tau \odot c$. The algorithm only keeps the expanded trails $\tau \odot c$ that pass the Accept predicate, thus forming the set of expanded trails $T_{\tau,\Gamma}$ (line 7). If the trail set $T_{\tau,\Gamma}$ is empty, then this means that τ could not be expanded further, so τ is union-grafted into the set of final trails F in line 8, which prevents its further consideration. Otherwise, the trail set $T_{\tau,\Gamma}$ must be union-grafted back into the trail set T. To ensure that each trail $\tau \in T$ refers to a unique method $sub(root(\tau))$, the algorithm must graft the trails whenever possible. The merging of $T_{\tau,\Gamma}$ and T is thus represented with the union-graft operation $\overset{\circledast}{\cup}$. The set H of hot subroutines is derived from T in line 14.

3.4 Hot-code-detection Policies

ALGORITHM 2: DetectHot Procedure

```
   input  : program call graph G, profile Π
   output : hot roots H, trail set T
1  T0 = InitialTrails(Π);
2  T = (fix ∇⊛_root)(T0);
3  F = ∅;
4  while ¬ DetectionDone(T, F) do
5      τ = TopTrail(T \ F);
6      Γ = CallingContexts(τ, Π, G);
7      T_{τ,Γ} = { τ ⊙ c : c ∈ Γ, Accept(τ ⊙ c) };
8      if T_{τ,Γ} = ∅ then  F = {τ} ⊛∪ F ;
9      else
10         T = T \ {τ};
11         T = T_{τ,Γ} ⊛∪ T;
12     end
13 end
14 H = { sub(root(τ)) : τ ∈ T };
```

The code in Algorithm 2 includes several procedures that drive the behavior of hot-code detection: InitialTrails, DetectionDone, TopTrail, CallingContexts, and Accept. A specific combination of implementations of these procedures is called a *hot-code-detection policy*. Different policies result in different instantiations of the algorithm, and we show our choice next. In this section, we explain the policy components, and we present the specifics of the key procedures in Appendix D.1 to facilitate the algorithm implementation.

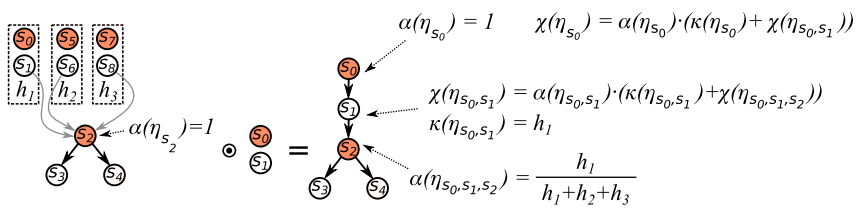
InitialTrails creates a set of initial trails from the profile Π. We use a constant ψ to set the lower bound for the profile's contribution to the total hotness of a program. The constant ψ is tuned on a set of benchmarks to achieve the best performance, as explained in Section 5.6. Next, DetectionDone is true once all trails from T are finalized. This is ensured by our choice of Accept.

CallingContexts procedure. The root of every trail represents the root subroutine of a compilation unit, which can be extended along a particular calling context. The set of all possible calling contexts can be determined by examining all the profile entries in the profiles Π. We identify the subset $\Pi|_c$ of profile entries in Π that refer to callsite executions specifically. For a specific trail τ,

the set *callerProfiles* is determined as those entries from $\Pi|_c$ that end with a subroutine s_n, which calls the subroutine $root(\tau)$ in the call-graph G.

The total amount of time spent in the code represented by the trail τ is the sum of the time spent when that trail τ is invoked from each of its calling contexts. When extending the trail τ, we would like to retain the extensions $\tau \odot c$ across those calling contexts c such that a significant portion of the time is spent in τ when called from c. Note that the information about the amount of time spent in a trail when called from a particular calling context is not included in the profile Π—the profile only includes the execution counts of *individual points in the program*. Therefore, we need to somehow estimate the portion of the trail τ's hotness that belongs to a particular extension $\tau \odot c$.

For this reason, the CallingContext procedure additionally computes an *attenuation factor* $a_c(\tau) \in [0, 1]$ of each context $c = \langle s_1, \ldots, s_n \rangle$, which is an estimate of the portion of τ's hotness that is the result of being called from the calling context c. The attenuation factor is calculated as a ratio of the hotness of one calling context c that calls $root(\tau)$ and the hotness sum of all such contexts. Importantly, this is just an estimate based on the partially context-sensitive profiles. The hotness counts h_c in the numerator and h in the denominator denote only *how many times the trail was called* and serve as a proxy for *the amount of time spent in those calls*. Consider the following example:



In the preceding figure, the trail $s_3 \leftarrow s_2 \rightarrow s_4$ (on the left) has the calling contexts $s_0 \rightarrow s_1$, $s_5 \rightarrow s_6$, and $s_7 \rightarrow s_8$, with hotness h_1, h_2, and h_3, respectively. The attenuation factors of these different calling contexts of the subroutine s_2 are computed as the ratio of a calling-context hotness and the sum of the hotness of all three calling contexts. Hotness of a trail is computed by summing up the hotness of its breadcrumbs, and weighting them using the attenuation factors.

Trail hotness. Hotter trails should generally have a greater likelihood of being considered for expansion and should get expanded across more calling contexts. The *trail hotness* is a function $\chi : \mathbb{T}(P) \rightarrow \mathbb{R}_0^+$ that maps a trail to a non-negative real value. It is defined as the recursive hotness sum of all the breadcrumbs in the trail τ, where each subtree is weighted with the graft-point attenuation α_τ Next, we explain the auxiliary functions α and κ.

Graft-point attenuation of a trail τ is a function $\alpha_\tau : N \rightarrow [0, 1]$ that maps each node to a real value between 0 and 1. The purpose of α_τ is to partially decrease the hotness of some parts of the trail when it gets extended along a calling context. For the initially created trails, the attenuation factor is set to 1 to all the graft-point nodes. After the grafting operation $\xi \circledast_\eta \rho$ is performed, the attenuation factor of the resulting trail graft points is inherited from the input trails. In other words, the attenuation factor of the root of the ξ trail, which is being grafted is set to the value of the graft point of the corresponding node of the target trail ρ. All the other attenuation factors in trail ξ are inherited from the input ξ trail. When a trail ρ is expanded using a calling context c, the algorithm computes the attenuation factor $a_c(\rho)$ for that particular extension in the CallingContext procedure, as previously described. This is the attenuation factor for the root of ρ trail, while the attenuation factor for the root of the expanded trail is set to 1.

Breadcrumb hotness of a trail $\tau = (N, E, \gamma)$ is a function $\kappa_\tau : N \rightarrow \mathbb{N}_0$, which maps each node to its estimated count. Initially, all the nodes corresponding to the end of a calling context have the hotness set to h from the profile entry the trail is constructed from. When expanding ρ trail

Table 3. Summary of Key Symbols from Section 3.4

Symbol	Name	Meaning
Π	Profile	Set of profile entries. Profile entry is a pair (L, h) of a calling context $L = \ell_1, \ldots, \ell_n$ and its execution count h.
$\Pi\|_c$	Callsite profiles	Subset of entries in Π that represent callsite executions.
ψ	Hotness threshold	Value above which a hotness entry is considered hot.
InitialTrails(Π)	Initial-trails heuristic	Creates an initial set of trails for a profile Π.
DetectionDone(T, F)	Termination heuristic	Determines if the hot-code detection must terminate.
callerProfiles(τ, Π, G)	Caller-profile set	Returns a set of profile entries from $\Pi\|_c$ that invoke the root subroutine of the trail τ in the call graph G.
CallingContexts(τ, Π, G)	Calling-context set	Returns the calling contexts in *callerProfiles*(τ, Π, G) (i.e., the same profile entries but without hotness).
$a_{\ell_1, \ldots, \ell_n}(\tau)$	Attenuation factor	Estimated hotness percentage of a trail τ when the trail τ is invoked from the calling context $\ell_1, \ldots, \ell_n$.
$\alpha_\tau(\eta)$	Graft-point attenuation	Maps each trail node η to an attenuation value in [0, 1].
$\kappa_\tau(\eta)$	Breadcrumb hotness	Maps each trail node η to the hotness of that node.
$\chi(\tau)$	Trail hotness	Estimated hotness of the entire trail τ.
$(\!\|T\|\!)_{hotter}$	Hotness-ordered trail set (Equation (18))	Sequence of trails from T, ordered by the $hotter$ relation.
TopTrail(T)	Top-trail heuristic	Returns the hottest trail from the set of trails T.
$\nu(\tau)$	Trail size	Code-size estimation for the trail τ.
$\zeta(\tau)$	Recursion penalty	Recursion penalty for the trail τ.
Accept(τ)	Acceptance heuristic	Decides whether to retain a trail τ after expansion.

with a calling context c, nodes that originate from ρ inherit the original hotness, and all the nodes corresponding to c have their hotness set to 0. When grafting is performed, the hotness of the grafted nodes is added together where possible and inherited otherwise.

TopTrail predicate. To decide which trails are most promising, the algorithm relies on the hotness of the individual profiles. The trail hotness χ induces a total order *hotter* on the set of all trails $\mathbb{T}$, where the tie-breaker is the lexicographic order of τ_1 and τ_2 (see Equation (14)):

$$\tau_1 <_{hotter} \tau_2 \equiv \chi(\tau_1) > \chi(\tau_2) \vee (\chi(\tau_1) = \chi(\tau_2) \wedge \tau_1 <_{lex} \tau_2)\,. \tag{18}$$

The TopTrail procedure then simply takes the first trail from T according to the *hot* order.

The Accept predicate. While our policy always works on the hottest trail, it is preferable to eventually process trails that are less hot, but smaller. For this reason, the Accept predicate must prevent further expansion of the hottest trail once the trail grows too large. To assess how large the trail is, we define a metric ν that sums the code sizes of all the subroutines in the trail.

It is useful to restrict the amount of recursion in the trail. To achieve this, we define the function ζ that computes the sum of $2^{recursionDepth}$ of all the nodes in the trail, multiplied by a small, experimentally determined constant k_{rec}.

Let the relative hotness be the hotness of the trail $\chi(\tau)$ divided by the approximation of the total time spent in the program (i.e., the sum of all the hotness counts from the profile Π). The extended trail is accepted if its relative hotness decremented by the recursion penalty $\zeta(\tau)$ is larger than the *threshold* function that depends on the size of the trail τ. Table 3 contains an overview of the key symbols and procedures from Section 3.4.

3.5 Inliner Modifications

After the hot-code detection from Algorithm 2 produces the trail set T that corresponds to hot compilation units, the compiler's inlining algorithm, which creates the individual compilation units, can exploit the information in the trail set T. Some inlining algorithms maintain a *call-tree*

data structure [33, 97, 114] (sometimes called *inlining plan* and *inlining tree*), while others work directly on the call graph [39, 96, 104, 109]. The decisions about expanding the compilation unit across the call graph or the call-tree data structure are often driven by cost-benefit analyses [33, 39, 45, 53, 62, 104].

While we speculate that the cost-benefit analysis of most inliners can be improved by using the hotness information in the trails, in this section, we demonstrate how we augmented a concrete inlining algorithm that is used in the Graal compiler [97]. This inliner maintains the *inlining tree* data structure, which is a *finite tree* nested within some call tree C, as defined by Equation (4).

$$\mathbb{I}(P) \equiv \{(N, E) : |N| < \aleph_0 \wedge E \subseteq N \times N \wedge \exists C \in \mathbb{C}(P), (N, E) \lessdot C\} \tag{19}$$

The inlining tree is (1) *expanded* until the inliner decides that a sufficiently large part of the call tree is covered. After that, the inliner (2) decides which parts of the inlining tree will be *inlined* into the compilation unit. Finally, the inliner (3) optimizes the inlining tree by pruning some branches, i.e., it attempts to find another inlining tree nested within the current one. These steps are repeated until a termination condition is satisfied—we summarize them in Algorithm 3.

During the expansion phase, the inliner repetitively selects a leaf node of the inlining tree and adds its child nodes. Algorithm 4 shows an individual inlining-tree expansion step—starting from the root, it keeps descending to the child with the highest priority φ until finding some node $\eta_{s_1,\ldots,s_n}$ that does not have children. If that leaf node does not have any callsites, then its priority φ is $-\infty$. Otherwise, φ of a leaf node is defined as its benefit divided by code size, where the benefit correlates with the call frequency. Priority φ of an inner node is equal to that of the child η_c with the largest priority φ, decreased by a penalty function that depends on the size of the respective subtree. To ensure that less frequent callees that are closer to the root compilation unit are not overlooked, the *penalty* term decreases the likelihood of exploring huge subtrees that have a high frequency [97]:

$$\varphi_{\textsc{GraalVM}}(\eta, \theta, G) \equiv \begin{cases} -\infty & \eta_{s_n} \rightarrow \eta_{s_{n+1}} \notin E_G \\ \mathit{benefit}(\eta) \big/ \mathit{codeSize}(\eta) & \eta_{s_1,\ldots,s_n,s_{n+1}} \notin N_\theta \wedge \eta_{s_n} \rightarrow \eta_{s_{n+1}} \in E_G \\ \max\limits_{\eta \rightarrow \eta_c \in E_\theta} \varphi_{\textsc{GraalVM}}(\eta_c, \theta, G) - \mathit{penalty}(\eta) & \mathit{otherwise} \end{cases}$$
$$\mathit{where} \quad \eta = \eta_{s_1,\ldots,s_n} \quad \theta = (N_\theta, E_\theta) \quad G = (N_G, E_G). \tag{20}$$

The modifications that we describe in this section apply to the InlineHot procedure from Algorithm 1. The InlineCold procedure corresponds to the unmodified version of the inliner.

ALGORITHM 3: GraalVM Inliner [97]

```
input  : program call graph G, entry-point e
output : IR u of the root compilation unit
θ = ({η_e}, ∅);
while ¬ InliningDone(G, θ) do
    θ = Expand(G, θ);
    θ = Inline(θ);
    θ = Optimize(θ);
end
η_u = root(N_θ, E_θ) where θ = (N_θ, E_θ);
u = GetIR(η_u);
```

ALGORITHM 4: Expand Step [97]

```
input  : call graph (N, E), inlining tree θ
output : expanded inlining tree θ
η = root(N, E) where θ = (N_θ, E_θ);
while {η_c : η → η_c ∈ E_θ} ≠ ∅ do
    η = arg max_{η→η_c∈E_θ} φ(η_c) y
end
η_{s_0,...,s_n} = η;
N_c = {η_{s_0,...,s_n,s_{n+1}} : η_{s_n} → η_{s_{n+1}} ∈ E};
E_c = {η_{s_0,...,s_n} → η_{s_0,...,s_n,s_{n+1}} : η_{s_n} → η_{s_{n+1}} ∈ E};
θ = θ ∪ (N_c, E_c);
```

Expansion modifications. The inliner has a limited budget for inlining-tree expansion, so inlining relies considerably on expanding the "most beneficial" parts of the inlining tree. We therefore bias the priority function φ so the inlining-tree nodes that correspond to nodes of some trail are

expanded with a higher likelihood. The rationale for this is that they are more likely to transitively invoke "hot code," which should be inlined into the root compilation unit. Our modification defines a new priority function $\varphi_{\text{InlineHot}}$, which multiplies the default priority with an experimentally determined constant k_{bonus} whenever the node in the inlining-tree can be matched to some trail:

$$\varphi_{\text{InlineHot}}(\eta_{s_1,\ldots,s_n},\theta,G) \equiv \varphi_{\text{GraalVM}}(\eta_{s_1,\ldots,s_n},\theta,G)\cdot\begin{cases} k_{bonus} & \exists(N_\tau,E_\tau,\gamma_\tau)\in T, \eta_{s_m,\ldots,s_n}\in N_\tau \\ 1 & otherwise \end{cases}. \tag{21}$$

Due to the *penalty* term in the priority function φ_{GraalVM}, the bonus merely biases the expansion around the "hot code" represented by the trails but does not completely prevent the exploration of the "cold" callees.

Budget modifications. When compiling hot compilation units, we change the parameters of the inliner to increase the code-size budget available for inlining. As described in the related work cited in Reference [97], the inliner uses the following threshold function that can prevent expansion, depending on the total code size ν of the entire inlining tree (as defined by Equation (35)):

$$benefit(\eta)/codeSize(\eta) \geq e^{\nu(\theta)/r}. \tag{22}$$

Next, the inliner uses the following threshold function to decide whether to inline a callee compilation unit η into the root compilation unit of the call tree θ, which is the caller of η:

$$benefit(\eta)/codeSize(\eta) \geq t_1 \cdot 2^{(codeSize(root(\theta))+codeSize(\eta))/t_2}. \tag{23}$$

We experimentally tune the parameters r, t_1, and t_2 to increase the size of the hot compilation units and to consequently improve performance on a set of programs, as described in Section 5.2.

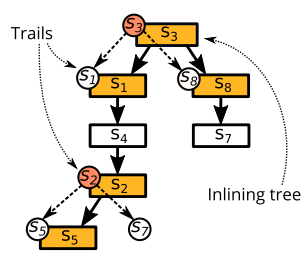
InlineCold modifications. The only change in the inlining for cold compilation units is that we prevent the inlining of subroutines in the set H from Algorithm 2, which were previously compiled as hot compilation units. The expansion priority of calls to subroutines $s_n \in H$ is set to $-\infty$:

$$\varphi_{\text{InlineCold}} \equiv \begin{cases} -\infty & s_n \in H \\ \varphi_{\text{GraalVM}}(\eta_{s_1,\ldots,s_n},\theta,G) & otherwise \end{cases}. \tag{24}$$

In addition, the Inline step from Algorithm 3 is modified to never inline a call to $s_n \in H$.

3.6 Hot-callee Classification

After the compiler inlines some of the callees into a hot subroutine, there will generally exist some remaining callsites in the corresponding compilation unit. Some of these callees may be "cold" and not beneficial for inlining, but some of them may be "hot" callees that were not inlined due to insufficient budget. Such "hot" callees should be recursively compiled as hot compilation units. The IsHot procedure from Algorithm 1 needs to decide whether a given callee of the compilation unit should be placed onto the hot or onto the cold compilation queue.



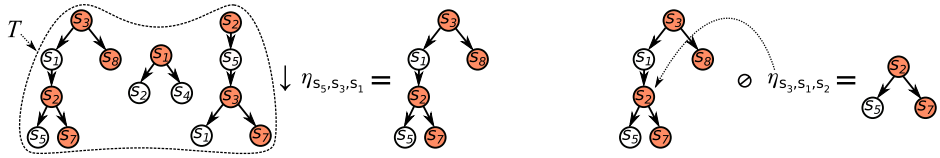
Consider the example in the adjacent figure, where an inlining tree is shown with rectangles, and the trails are shown with circles. The trail $s_1 \leftarrow s_3 \rightarrow s_8$ is matched against the root of the inlining tree, and the corresponding nodes of the inlining tree are shaded in yellow. The non-matched (cold) parts of the inlining tree are shaded in white—the node s_4 is not matched to any node in the trail, but its callee s_2 can be matched to another trail $s_5 \leftarrow s_2 \rightarrow s_7$. We classify such callees as hot—for example, if s_5 does not get inlined, then it will be recursively compiled as hot.

Table 4. Summary of Key Symbols from Sections 3.5 and 3.6

Symbol	Name	Meaning
$\mathbb{I}(P)$	Inlining tree (Equation (19))	Data structure maintained by the inliner for the program P.
φ_{GraalVM}	GraalVM node priority function (Equation (20))	Function that calculates the node priority using heuristics in GraalVM [97].
$\varphi_{\text{InlineHot}}$	InlineHot node priority function (Equation (21))	Function that increases an inlining-tree-node priority if it is marked as hot.
$\varphi_{\text{InlineCold}}$	InlineCold node priority function (Equation (24))	Function that modifies an inlining-tree-node priority if it is marked as cold.
$T \downarrow \eta_{s_1,\ldots,s_m,\ldots,s_n}$	Trail-matching operation	Operation that finds a trail in the trail set T containing the longest suffix of the call sequence $\eta_{s_1,\ldots,s_m,\ldots,s_n}$.
$\tau \oslash \eta_{s_1,\ldots,s_m}$	Trail-cut operation	Operation that produces a new trail that contains only a subtree of the τ trail starting from the node $\eta_{s_1,\ldots,s_m}$.

The trail for the hot callee $\eta_{s_1,\ldots,s_n}$ is determined with a combination of trail matching and trail cutting operations. First, we find a matching trail for $\eta_{s_1,\ldots,s_n}$ in the trail-set T, and we then cut that trail at the subtree that corresponds to the call sequence $s_1, \ldots, s_n$. The *trail-matching operation* $\downarrow$ matches a node from the inlining tree, with a call sequence $s_1, \ldots, s_m, \ldots, s_n$, to a trail τ from the trail set T, such that τ contains the longest suffix of that call sequence. For the initial set of hot subroutines H from Algorithm 2, the association is straightforward, since each $s \in H$ corresponds to a trail $\tau \in T$. However, a hot call to a subroutine s may not have a corresponding trail—for this reason, we implemented a trail-cutting operation. Given a breadcrumb trail τ and its node $\eta_{s_1,\ldots,s_m}$, the *trail-cut operation* $\oslash$ produces a new trail that consists only of the subtree starting at $\eta_{s_1,\ldots,s_m}$. Definitions and details of both operations are presented in Appendix D.2.

The trail-matching and trail-cutting are illustrated in the following example. The longest trail-match in the set T for the calling context $\langle s_5, s_3, s_1 \rangle$ is the leftmost trail in the figure, because that trail contains the longest suffix of s_5, s_3, s_1. The resulting trail is then cut on the right side of the figure, at node η_{s_3,s_1,s_2}, and this yields the trail $s_5 \leftarrow s_2 \rightarrow s_7$. Both operations as well as other key symbols from Sections 3.5 and 3.6 are presented in Table 4.



3.7 Parameter Summary

In Table 5, we listed the key parameters ψ, k_{bonus}, and k_{rec}, which the proposed algorithm introduces. Constant ψ is a limit that a profile's hotness must exceed to be included in the initial set of hot profiles, compared to the overall hotness of the program. Constant k_{rec} serves as a multiplier for calculating the recursion penalty of a trail. The details of how we used these constants in the algorithm's policy can be found in Appendix D.1. k_{bonus} is a constant for multiplying a default inlining-tree-node priority for every node that can be matched to a trail, according to Equation (21). All these constants are tuned for the best performance on a set of benchmarks, and the results are presented in Section 5.6. Notably, since different compilers and inliners have different IR designs and enact different optimizations, their cost and benefit models differ, so we expect that the algorithm parameters need to be re-tuned for each specific environment.

Table 5. Summary of the Key Introduced Parameters

Symbol	Name	Meaning
ψ	Hotness threshold	Value above which a hotness entry is considered hot.
k_{rec}	Recursion constant	Constant used for restricting the amount of recursion in a trail.
k_{bonus}	Expansion bonus (Equation (21))	Bonus for expanding an inlining-tree node matched to a trail.

Table 6. Summary of the Existing Inliner Parameters in GraalVM

Symbol	Name	Meaning
r	Expansion inertia base (Equation (22))	Parameter that impacts the amount of call-tree expansion, which the inliner performs.
t_1	Relative benefit coefficient (Equation (23))	Parameter driving the benefit threshold for deciding whether a method should be inlined.
t_2	Base target spending (Equation (23))	Parameter limiting the budget that the inliner has available for inlining.

This section also contains the key parameters already included in the inliner, and they are listed in Table 6. We tuned them to show that our algorithm does not show better performance only for a subset of the parameter values. As a result of the experiments, we chose the values that give us good performance, but we also demonstrate that our algorithm works well for the other parameter values. These results are presented in Section 5.2.

4 IMPLEMENTATION

The algorithm presented in Section 3 was described on an abstract level, and it can be implemented in most concrete compilers. To evaluate the algorithm, we implemented it inside the ahead-of-time Graal compiler for the GraalVM Native Image [120] and applied it to concrete programs and benchmarks. More precisely, we modified the compile queue within the existing compiler, added a phase to analyze the input profiles, and used it to improve the decisions of the existing inliner.

We start with an overview of the existing compiler inside GraalVM and the existing profile-guided optimization support in GraalVM's ahead-of-time compilation mode in Sections 4.1 and 4.2. Then, in Sections 4.3 and 4.4, we discuss the data structures and the details of our implementation. If the reader wishes to see an example execution of the proposed algorithm, Appendix A illustrates compilation of a method from MNEMONICS benchmark.

4.1 System Overview

AOT compilation is a set of techniques for compiling and optimizing the entire program before the execution of the program begins. GraalVM Native Image [120] is an ahead-of-time compiler in GraalVM, in which we implemented the algorithm that is the subject of this article. GraalVM Native Image compiles the input program to a platform-specific executable, called **native image (NI)**. The input to the Native Image is the set of class-files that contain the Java bytecode [1] that represents the program and the name of the method that is the program entry point.

JIT is a substantially different compilation paradigm, in which the compilation is performed during the execution of the program, and is done only selectively on a subset of methods. The execution of the program usually starts in the first stage of the compiler, e.g., an interpreter. After the method gets invoked a certain number of times, or after the total amount of the time spent in the method exceed a specific threshold, that method is placed on the compilation queue and eventually compiled. HotSpot [95] is one of the virtual machines that uses the JIT compilation approach.

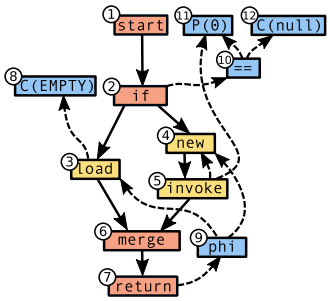


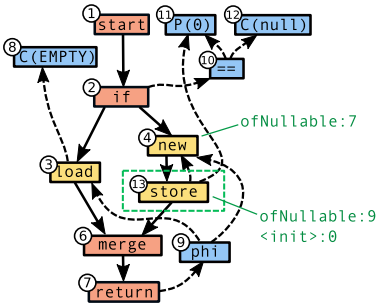
Fig. 4. Example graal IR for the `Optional.ofNullable` method.

In the GraalVM ecosystem, both Native Image and HotSpot use Graal as the optimizing compiler. After the Graal compiler loads the bytecodes of the program, parses them, and creates the corresponding **intermediate representation (IR)** [67], it applies numerous optimization phases to the program IR. Graal is, in principle, an intraprocedural optimizing compiler—every method of the program is parsed as a separate compilation unit, and all subsequent optimization phases are separately applied to each compilation unit. However, a single compilation unit may comprise several methods, because the compiler can decide to inline some of the callees.

Graal IR. The intermediate representation in the Graal compiler is a directed graph data structure that simultaneously models the control-flow and dataflow dependencies between individual execution steps [67], similar to the sea-of-nodes representation [58]. Figure 4 shows the corresponding Graal IR for the JDK 8 `Optional.ofNullable` method [18]. The method is defined as follows:

```
<T> Optional<T> ofNullable(T x) { return x == null ? EMPTY : new Optional<>(x); }
```

Inlining and node source positions. The Graal compiler transforms the aforementioned IR in a sequence of transformation phases, one of which performs inlining [97]. In the previous `ofNullable` example, the `invoke` node represented the constructor call, which merely assigns the parameter to one of the fields of the `Optional` object. Typical heuristics inline this constructor due to its small size [96], and the resulting state of the IR is shown in the following figure.



The `invoke` node that represents the `Optional` constructor call was replaced with the body of the constructor, which is in this case a single `store` node. We use this example to explain the concept of a *node source position* in Graal. The Java bytecode of each method is a linear list of instructions [75], each of which has a unique ***bytecode index (BCI)***, which is the offset of the instruction in bytes. For a particular method, BCIs start from zero and go up to the BCI of the last instruction. Every node in the IR is associated with the bytecode instruction that it was created from. Therefore, every node has a method-BCI pair that denotes where it came from. However, nodes can originate from callees that were inlined into the compilation unit, and for this reason, Graal assigns a *list* of method-BCI pairs to each node, which represents where the node was inlined from. This list is called the *node source position*. In the preceding example, the node source position of the `new` node is `ofNullable:7`, because this particular node was not inlined (i.e., it comes from the root method of the compilation unit). The node source position of the `store` node is `ofNullable:9,<init>:0`, since that node was inlined from the call at `ofNullable:9`, and is the first instruction in the constructor `<init>` of the `Optional` class.

The *calling context* is the node source position of the `invoke` from which a particular node is inlined. For the `store` node in the example, `ofNullable:9` is its calling context. The term calling context is overloaded, because it also denotes the call stack that exists during the program execution and which invokes the compilation unit that corresponds to the top frame of the call stack—to disambiguate, we say *call stack* when we mean the latter.

The build process in Native Image. Image-building consists of several main steps: (1) points–to-analysis, which includes class initialization and heap snapshotting, (2) compilation, and (3) image-heap writing [120]. The points-to analysis step determines the set of reachable classes and methods [113], which is called the *hosted universe*. The points-to analysis also initializes the static fields of some classes. The compilation step then, starting from the entry points, recursively compiles the methods from the hosted universe one-by-one in a manner that corresponds precisely to lines 13–20 in Algorithm 1. Finally, the image-writing step creates a binary with the compiled code and the initial state of the heap (which comprises the objects that are transitively reachable from the static fields of the initialized classes).

Extensions in this work: the compilation is separated into two rounds, the first compiling hot methods and the second compiling cold methods.

4.2 Overview of Profile-Guided Optimizations (PGO) in Native Image

On HotSpot, the program is profiled while it is interpreted and before it is JIT-compiled [15, 50, 80, 122]. Since method inlining (which by definition introduces calling-context sensitivity into the compilation unit) is only performed during JIT compilation, the profiles collected by the interpreter are not context-sensitive—the interpreter maintains the same set of profile counters for a particular method, regardless of who the caller is. In AOT compilation, however, the entire program is compiled before the program starts executing, and there is typically no interpreter available. For this reason, GraalVM Native Image supports two modes of compilation—*instrumentation image*, in which the binary is instrumented with the code that collects the profiles, and the *optimization image*, in which the binary does not have any profiling code. The program is first compiled into the instrumentation image, which is executed once to collect the profile information. The profile information is dumped to a file when the instrumentation-image program exits. Then, the program is compiled into the optimization image, which optionally takes the file with the profile information. This profile information is then used to guide the decisions in various compiler optimizations.

Types of profiled events. Specific instructions of interest in the profiled program are called *events*. To minimize the performance overhead of profiling, most VMs track only those events that are considered useful in improving the effectiveness of compiler optimizations. GraalVM Native image collects profile information for three different types of events:

- **Method entries.** Records the number of times that a method M is entered. The method M may be inlined into a calling context C, in which case this event type represents method's hotness within the calling context C.
- **Conditional branches.** Records the number of times that each branch of a conditional (an `if` or a `switch` statement) was executed. Each branch of the conditional is associated with a BCI within the method M, which may be inlined into a calling context C.
- **Virtual dispatches.** Keeps an array of counters, each for one potential receiver type at the virtual callsite, and records the number of times that each concrete receiver type was invoked. The callsite is associated with a BCI in M, which may be inlined into a calling context C.

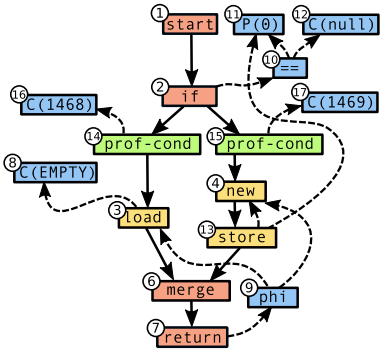
Extensions in this work: The virtual-dispatch profile for each callsite in hot code is restricted to the callees that appear on the breadcrumb trail instead of being taken directly from the input profile.

Obtaining profiles. In Native Image, the profiling is based on instrumentation—the IR of each compilation unit is modified to include nodes that collect counts for the previously enumerated events. Each node that is inserted into the IR represents a sequence of instructions that updates the

counter for the event associated with the corresponding node source position. Each node source position of a profiling node is mapped to a unique identifier, which is used as an index in a counter array. Since the node source position includes the calling context within the current compilation unit, the count associated with every event is context-sensitive.

> *Extensions in this work: Partially context-sensitive profiles are effectively made longer using the proposed breadcrumb trails approximation.*

The Native Image developers decided not to use atomic counters in Native Image PGO. The rationale behind this decision was twofold: First, the extensive measurements revealed that the errors introduced by non-atomic counters are small, while the overhead of atomic counters is considerable compared to non-atomic ones; and, second, optimization decisions that are made based on counter values are in most cases heuristics and thus approximate. Furthermore, non-atomic counters in Native Image may lose some updates but cannot lose all of them—every calling context that was executed in the instrumentation image always has a non-zero profile count. The associated calling contexts contain all the methods from the program traces. Finally, our goal was to compare our proposed technique against the existing PGO in Native Image without changing the instrumentation mechanism, so modifying the existing counter infrastructure was out of scope.



The adjacent example shows the IR after the insertion of two `prof-cond` nodes (one for each branch following the `if` node 2), which represent increments of the counters 1,468 and 1,469, respectively. In the later compilation phases, these nodes are replaced with low-level memory loads and stores that increment the respective counter. The profiling nodes for event types such as virtual dispatches also count the corresponding receiver types and are lowered to code that maps the receiver type to its entry. Once the instrumentation image of the program completes execution, the values of these counters are dumped to the disk into a file, which is consumed by the optimization image.

The profile format corresponds to the formal input of the algorithm, as described in Section 3. Each counter is associated with the node source position of the corresponding instrumented node. The first location in the node source position always corresponds to the root method of the compilation unit, and the last location is the method-BCI pair of the instruction that the instrumented node was parsed from.

One of the main reasons that the Native Image implementers decided to use only partially context-sensitive profiles is that the collection has a relatively low performance overhead and that the memory consumption of the counters is low. While this instrumentation contains the exact number of times that an event was executed in a given partial calling context, its downside is that the partially contextual profiles can be polluted in compilation units that have many callers, because the profile counts are aggregated across all callers. This, in turn, leads to incorrect inlining decisions. The length of partial calling contexts can be increased by doing more inlining in the instrumentation image, but this "works" only to a certain degree, as we show in Section 5.5.

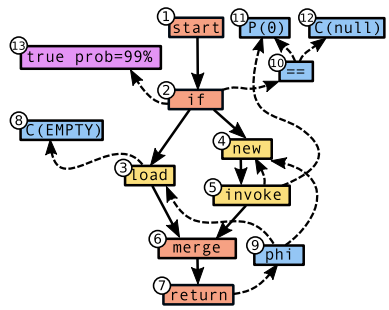
Applying profiles. Native Image already relied on PGO prior to our work in the following manner: The nodes in the Graal IR are augmented with additional information, such as the probability that a particular branch is taken at a particular `if` node or the probability of a particular receiver type appears at a particular `invoke` node. This information is calculated from the counts in the profiles and (when present) is used to guide existing Graal optimizations with

knowledge about the program's execution. For example, branch probabilities are used to compute basic block frequencies that guide path duplication [88] and loop transformations, and receiver type probabilities are used in callsite devirtualization [78, 97]. In this section, we explain how the partially context-sensitive profiles are applied to the IR in Native Image.

The **bytecode-parsing phase**, which constructs the IR of each method in the native-image program, is the first phase in which the profiles are applied. During bytecode parsing, the profiles are applied in a *context-insensitive manner*, because the IR of a particular method is parsed, and no inlining occurred so far. Inlining, a later phase in the pipeline, applies the profiles in a context-sensitive manner, with the purpose of helping the inliner make better decisions.

The two main steps of the **inlining phase** in GraalVM, as explained in Section 3.5, are expansion and inlining. Native Image applies the *context-sensitive profiles* during the expansion step of the inliner—in each of the callee methods, the branch probabilities of all `if` nodes and the receiver-type probabilities of all `invoke` nodes are computed using only those profile entries that match the calling context of the respective callee in the inlining tree, as explained next.

Profile-application example. Consider the adjacent figure, where the profiles from the instrumentation run were applied to the compilation unit `ofNullable` of the optimization image. The purple node 13 denotes the probability of taking the `true` branch at the `if` node 2, which can be utilized by various optimizations. For example, the inlining phase can now conclude that it is very unlikely that the constructor call at the `invoke` node 5 in the `false` branch will be called, so there is very little benefit from inlining it. However, subsequent code-motion phases can decide to move the highly frequent `load` node 3 to the position before the `if` node 2, since speculative prefetching can result in a performance improvement [65, 100].



Querying profiles. The optimization-image compilation needs to map node source positions to execution counts. When the calling contexts within each compilation unit of the instrumentation image exactly correspond to the calling contexts of the optimization image, this mapping is trivial. However, the calling contexts generally differ between the instrumentation and the optimization image, due to different inlining decisions—the reason for this is that the inlining decisions are driven by the profiling information.

In Native Image, when querying the execution counts for the (shorter) node source position $\ell_1, \ldots, \ell_n$, the rule is to add together the counts associated with *all* (longer or equal) node source positions $\ell_1, \ldots, \ell_n, \ldots, \ell_m$ (where m may be equal to n, i.e., there is an exact match). The rationale for this is that all longer or equal calling contexts from the instrumentation image may correspond to the queried calling context. However, when a node source position $\ell_1, \ldots, \ell_n$ does not have an equal or longer match in the set of profiles, in Native Image, the rule is to add together all the profiles with shorter calling contexts.

Notably, these two rules were chosen as approximations by the Native Image developers. Generally, global analyses of the call graph and the profile set can result in better approximation schemes, but that is outside of the scope of this work—here, we outline how the existing profiling system works and do not change the default profile-querying schemes when implementing our algorithm.

Implementation inputs. The input to our implementation is the hosted universe that is produced by the points-to analysis step of the Native Image build (as explained in Section 4.1), represented with the `HostedUniverse` class in Native Image; the set of entry-point methods of the program, each represented by the `HostedMethod` class; and the set of pairs consisting of a

node-source-position and an integer (i.e., the PGO profile). This input closely corresponds to the input of Algorithm 1 from Section 3—notably, the hosted universe contains the call graph defined in Section 3.1, along with the set of reachable classes and their fields.

4.3 Data Structures for the Code Analysis

The previous two sections gave an overview of GraalVM, Native Image, and the existing profile-guided optimizations within Native Image. In the rest of the sections, we focus on our contribution and present our implementation of the new techniques proposed in the article. In this section, we present the implementation of the main data structures that we use for the code analysis. To explain the implementation, written in the Java programming language, we define the main fields of each of the data structures, and we show their most important methods.

```
class Breadcrumb {
    HostedMethod method;
    Map<Integer, Breadcrumb[]> callsites;
    Breadcrumb parent;
    GraftPoint graftPoint;
}
class GraftPoint {
    double attenuation;
}
```

Listing 3. Breadcrumb Data Type.

Data type `Breadcrumb`. Listing 3 shows the implementation of a breadcrumb, i.e., a node in an annotated breadcrumb trail as defined by Equation (7). The node corresponds to a single subroutine in the program (stored in the field `method`). It contains a reference to a parent node (field `parent`), which represents the caller of the node's subroutine in a given trail. The `callsites` map matches a bytecode index of each callsite in the node's method to the list of the nodes corresponding to the methods that can be invoked at the callsite. The rationale for having the list is that each bytecode index may represent a virtual call, which can dispatch to more than one method. Each breadcrumb has a non-`null` `GraftPoint` object if that breadcrumb represents a point for a grafting operation, as explained in Section 3.2. The `GraftPoint` object contains the `attenuation` field, which models the *graft-point attenuation* function of the associated trail, as formally explained in the definition in the Appendix D.1.

Data type `Trail`. In Listing 4, we present the concrete implementation of a breadcrumb-trail data type, which was formally defined by Equation (7) for annotated trails, in Section 3.2.

A trail is a tree that consists of breadcrumb nodes. The root breadcrumb (field `root`) corresponds to the method of the corresponding compilation unit. Every other breadcrumb has exactly one parent and the arbitrary number of children (Listing 3). A trail contains a mapping `grafts` from methods to all the breadcrumbs in the trail that correspond to the respective method and that have `GraftPoint` objects. This map is used to optimize the trail-grafting operation defined by Equation (9). Finally, to model various trail-related functions defined in Section 3, a `Metrics` object is associated with each trail. This object contains the total `hotness` field, which represents the hotness function $\chi(\tau)$ from Equation (34); the `size` field, which represents the $\nu(\tau)$ size function from Equation (35); and the `recursion` field, which maps each method to its greatest recursive depth in the respective trail and which is used to implement the penalty $\zeta(\tau)$ from Equation (36).

Listing 4 also contains methods that implement the three main operations defined on trails. These operations are already formally defined in Section 3.2. Method `createInitial` conceptually corresponds to the operation $(\varnothing, \varnothing, \varnothing) \odot \langle \ell_1, \ldots, \ell_n \rangle$ of extending an empty trail with a calling context, as defined by Equation (8). It takes a `Profile` object (consisting of a calling-context and the corresponding count `value`) as an input; its output is a new trail object, in which each breadcrumb coincides with a location from the input calling-context. The profile count is used for the initial hotness metrics of the created trail. Trail size and recursion depth are computed from the context input. The root of the resulting trail is a new graft point with the `attenuation` 1.0.

Method `expand` performs the expansion operation on `this` trail object and implements the $\odot$ operation from Equation (8). The `profile` argument of the method contains the context for

expansion and the profile count of that context. For each location in the context, we create a new breadcrumb and update `grafts` similarly to the construction of a new trail.

The `graft` method implements the grafting operation $\circledast$ from Equation (9)—it grafts the input trail (`graftee`) onto `this` trail. The grafting can only occur on those graft-point breadcrumbs from the target trail `this` whose method corresponds to the root method of the `graftee` trail—the `grafts` map allows efficiently retrieving this subset. For this reason, the grafting of the same `graftee` can occur more than once within a single trail. For each graft point, the `graftee` trail is copied and attached to the respective graft point. The metrics of the resulting trail are updated with the metrics of the grafted trail. The total hotness of the grafted trail is complemented with the hotness of the `graftee` multiplied with the `attenuation` of the graft point where the grafting happened, as per Equation (34). The remaining two metrics, `size` and `recursion`, are updated in a similar manner that ensures the Equations (35) and (36) are satisfied.

```
class Trail {
    Breadcrumb root;
    Map<HostedMethod, Breadcrumb[]> grafts;
    Metrics metrics;

    static Trail createInitial(Profile profile);
    Trail expand(Profile profile);
    Trail graft(Trail graftee);
}
class Metrics {
    long hotness;
    int size;
    Map<HostedMethod, Integer> recursion;
}
class Profile {
    Location[] context;
    long value;
}
class Location {
    HostedMethod method;
    int bci;
}
```

Listing 4. `Trail` Data Type.

The `TrailSet` data structure. During the execution of the algorithm in Listing 2, the DetectionDone call in line 4, the TopTrail call in line 5, the addition of the current trail τ to the finalized set F in line 8, removal of the current trail τ from the set T in line 10, and the union-graft operation in line 11 are executed multiple times. Using a naive set encoding, each of these operations would take $O(n)$ computational steps, where n is the total number of trails in the set. To reduce their computational overhead, the implementation uses the `TrailSet` data structure shown in Listing 5.

```
class TrailSet {
    PriorityQueue<Trail> queue;
    HashSet<Trail> finalized;
    HashMap<HostedMethod, Set<Trail>> candidates;

    boolean isFinalized();
    Trail popHottest();
    int unionGraft(Trail graftee);
    void graftToRoots(Trail trail);
}
```

Listing 5. `TrailSet` Data Type.

The `TrailSet` contains a priority queue, a hash-set of `finalized` trails, and a `candidates` hash-map that maps each method to a set of trails that contain that method in at least one graft-point. The non-finalized trails are in the queue, and finalized trails are in the `finalized` hash-set, so in the notation of Listing 2, $T \equiv$ queue $\cup$ `finalized` and $F \equiv$ `finalized`.

The `TrailSet` data structure includes several methods. The `isFinalized` method checks if the queue is empty, which effectively executes the DetectionDone policy from Equation (31) in $O(1)$ time. The `popHottest` method uses the queue to extract the hottest trail in $O(\log n)$ time and is used to implement the TopTrail policy (the ordering is determined by the `hotness` metric of the trails, as per Equation (34)). The `candidates` allows finding all candidate graft points in all trails in $O(C)$ time, where C is the size of the resulting set, so the overall computational time of union-grafting $\overset{\circledast}{\cup}$ (as defined by Equation (15)) only depends on the size of the resulting trails. The `finalized` hash-set is used to extract the final set of trails (set H from Listing 2).

4.4 Implementation Details

The preceding sections showed how the implementation of the algorithm in Native Image corresponds to the formalization from Section 3 but left out details such as how the hotness of a calling context is computed and how the set of calling contexts is determined (i.e., the CallingContexts policy). We now show how our implementation computes the hotness of virtual and direct callsites and how it estimates the hotness value of the individual breadcrumbs. Finally, we show how to determine the calling contexts and the attenuation values of each method.

Determining the hotness of virtual callsites. For each calling context ending with a virtual call, Native Image PGO data maps every concrete method to the invocation count of that method:

$$\pi(\ell_1 \dots \ell_n) = \{(s, N) : s \in \mathit{implementations}(\mathit{target}(\ell_n))\}. \tag{25}$$

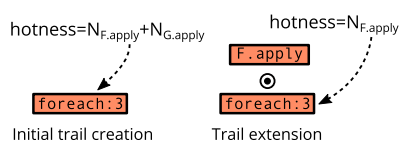
Above, the $\mathit{target}(\ell_n)$ is the base method for the dispatch at the virtual callsite ℓ_n, and *implementations* is a function that returns the set of methods that implement the base method. This profile entry has two interpretations, depending on when it is used. When creating a set of initial trails, a breadcrumb that gets created from a virtual call has its hotness increased by the sum of the invocation counts of all the possible concrete methods (because all of them contribute to the hotness of the trail):

$$h_{\ell_1,\dots,\ell_n} = \sum_{(s,N)\in\pi(\ell_1\dots\ell_n)} N. \tag{26}$$

When extending a breadcrumb trail rooted at a subroutine s with a calling context that ends with a virtual call, the invocation count that contributes to the hotness is the count for the method s:

$$h_{\ell_1,\dots,\ell_n|s} = N \quad \mathit{such\ that} \quad (s, N) \in \pi(\ell_1, \dots, \ell_n). \tag{27}$$

Thus, when creating the initial set of trails in line 1 of Listing 2, Equation (26) determines the hotness of the callsite that is a virtual call. When extending the top trail in line 7 of Listing 2, Equation (27) determines the hotness of the callsite that represents a virtual call.

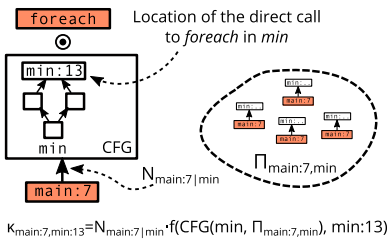


Example. To illustrate the preceding equations, consider the running example from Listing 1 and its calling context `foreach:3` in the adjacent figure, which ends with a virtual call to `apply`. The `apply` is a base method, and its concrete implementations are `F.apply` and `G.apply`. If this context is used to create an initial trail, then both the `F.apply` and the `G.apply` invocation counts contribute to hotness. However, if a trail that consists of a single breadcrumb `F.apply` is extended with its calling context `foreach:3`, then only `F.apply`'s invocation count contributes to hotness, because `G.apply` is not invoked in the extended trail.

Determining the hotness of direct callsites. Unlike the virtual callsites, the direct callsites do not have the invocation-count records in Native Image PGO. The hotness of a direct call at location ℓ_n in the calling context $\ell_1, \dots, \ell_{n-1}, \ell_n$ must be computed from the branch probabilities of the subroutine s_n that corresponds to the last location ℓ_n of the calling context. To do this, we first extract the set $\Pi_{\ell_1,\dots,\ell_{n-1},s_n|B}$ of the profile entries $\ell_1, \dots, \ell_{n-1}, \ell'_n$ that end with a location ℓ'_n that is inside the subroutine s_n, such that ℓ'_n is the location of a branch instruction. We then use the `ControlFlowGraph` class of the Graal compiler [13] to create a control-flow graph of the subroutine s_n, in which the basic block s_n-relative frequencies are computed using the branch execution counts from $\Pi_{s_n|B}$. The s_n-relative frequency of the basic block that contains the direct call ℓ_n is then multiplied with the invocation count of s_n to obtain the count in the respective calling-context:

$$h_{\ell_1,\dots,\ell_{n-1},\ell_n|s} = N_{\ell_1,\dots,\ell_{n-1}|s_n} \cdot f(CFG(s_n, \Pi_{\ell_1,\dots,\ell_{n-1},s_n|B}), \ell_n). \tag{28}$$

Above, *CFG* returns the control-flow graph of the subroutine s_n for the branch profiles $\Pi_{s_n|B}$, f retrieves the s_n-entry-point-relative frequency of the basic block that contains the location ℓ_n in the given control-flow graph, and $N_{\ell_1,\ldots,\ell_{n-1}|s_n}$ is the number of times that the method s_n is entered from the calling context $\ell_1,\ldots,\ell_{n-1}$, which ends with a call to s_n. This hotness value is used both when creating an initial set of trails and when extending existing trails.



Example. To illustrate Equation (28), consider the running example from Listing 1 again and its calling context `main:7,min:13`, which ends with a direct call to `foreach`. To compute the hotness of the direct call to `foreach` at `main:7,min:13`, we extract the set of branch profiles $\Pi_{\texttt{main:7,min}|B}$ that correspond to calling contexts of the form `main:7,min:X`, where `min:X` is any location in `min`. The branch profiles in $\Pi_{\texttt{main:7,min}|B}$ are then used to create a control-flow graph of `min` and a mapping from basic blocks to their frequencies relative to the entry point to `min`. We use this mapping to obtain the `min`-relative frequency $f(CFG(\texttt{min}, \Pi_{\texttt{main:7,min}|B}), \texttt{min:13})$ of the basic block that contains the location `min:13` of the direct call and multiply it with the number of times $N_{\texttt{main:7|min}}$ that the method `min` was entered from `main:7`. The result is the hotness of the direct call at `main:7,min:13`.

Estimating the attenuation of a calling context. As we have just previously shown, in our implementation, direct calls are not represented in the profiles, so the attenuation-factor calculation, defined in Equation (33), must be modified to include those direct calls in the denominator:

$$a_{\ell_1,\ldots,\ell_n}(\tau) \equiv \frac{h_{\ell_1,\ldots,\ell_n|\tau}}{\sum_{L\in directCallers(\tau)} h_{L|\tau} + \sum_{(L,h_{L|\tau})\in callerProfiles(\tau)} h_{L|\tau}}. \tag{29}$$

Above, the denominator is the sum of the hotnesses $h_{L|\tau}$ of contexts L that directly call τ (as per Equation (28)) and the hotnesses $h_{L|\tau}$ of contexts L that indirectly call τ (as per Equation (27)). The direct-call hotness and attenuation-factor calculations are cached to decrease the overhead.

Determining the calling contexts. To compute the set $directCallers(\tau)$, our implementation preprocesses the IR of the methods in the `HostedUniverse` and creates a mapping from each method to the list of its callsites—this callee-to-callsite table contains both the direct and the virtual callsites. The mapping is also used to implement the CallingContexts policy from Section 3.4.

5 EVALUATION

The main objective of the evaluation is to compare the performance of the proposed ahead-of-time inlining and compilation-scheduling algorithm with the existing state-of-the-art inliner that is used in GraalVM [97] (Section 5.1). To demonstrate that the comparison is fair, we inspect a range of parameters that affect inlining—for both algorithms, we find the parameter values that give the best possible performance (Section 5.2). Furthermore, the evaluation shows that the new inlining algorithm achieves improved performance with a minimal compiled-code size increase (Section 5.3).

To characterize how the different components of the algorithm affect peak performance of compiled programs, we then analyze the impact of those different algorithm components. In Section 5.5, we evaluate how the average length of the partial contexts correlates with the peak performance of compiled programs. Then, in Section 5.6, we show how heuristics such as the *inlining-budget boost* and the *trail expansion* formally defined in Section 3.2 affect the peak performance and how we tuned the threshold for the initial set of hot contexts.

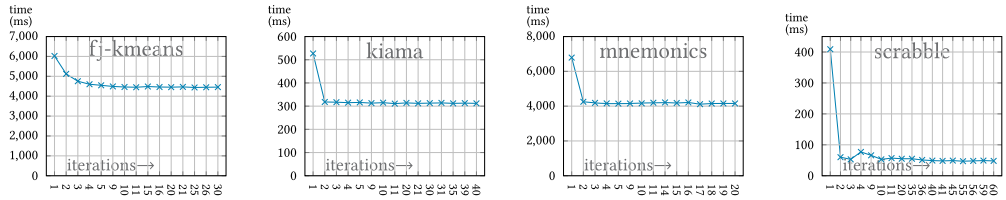


Fig. 5. Warmup curves on native image.

In Section 5.7, we explain the performance differences between ahead-of-time and just-in-time compilation with GraalVM, and we demonstrate that some of the differences are not due to the inlining decisions. Appendix B contains a case-study in which we explain how our proposed inlining algorithm deals with the problems that were described in Section 2 on the example of a flame graph of the *mnemonics* benchmark.

Experimental setup and methodology. Benchmarking was conducted on an Intel Xeon E5-2699v3 CPU with 18 cores and hyperthreading, with 264 GB main memory and tmpfs file system, running Oracle Linux Server release 6.10. During the experiments, turbo boost was disabled and we set the frequency of all CPU cores to 2.3 GHz to eliminate the effects of dynamic frequency scaling. All experiments were conducted as follows: For each data point, we did five separate measurements in different process instances, running different GraalVM Native Image (Native Image 20.3.0-dev (Java 8, revision: 7955c628b5c) with the default garbage collector) configurations, or JVM (depending on the specific experiment). Within each process instance, we repeatedly executed the benchmark for a predefined number of repetitions N, where N was selected on a per-benchmark basis to ensure that the steady state is reached before 60% repetitions are completed. We considered the benchmark steady once it ran for at least 20 seconds and after the coefficient of variance reaches a threshold [74]. Warmup time for GraalVM Native Image is shorter than the warmup time on JVM. There are only several initialization steps of the image heap executed at runtime before the main method of the benchmark is invoked [120]. Figure 5 demonstrates that a benchmark's running time becomes stable after only a couple of iterations. The number of repetitions of each benchmark for Native Image and JVM is shown in Table 7. We then computed the average execution times across the last 40% repetitions of all five measurements. In the results, we also show the standard deviation, except when presenting the impact on the code size (where the results were stable and the standard deviation was insignificant).

Table 7. Number of Iterations per Benchmark

Benchmark	NI Iterations	JVM Iterations
h2	10	16
fj-kmeans	20	40
mnemonics	20	60
par-mnemonics	20	60
philosophers	15	30
reactors	10	10
rx-scrabble	80	150
scala-stm-bench7	30	60
scrabble	60	250
apparat	10	20
kiama	40	120
scalac	40	120
scaladoc	40	120
scalap	120	250
scalariform	40	120
tmt	12	16

Workloads. We used 16 benchmarks from DaCapo [46], Scalabench [108], and Renaissance [99] benchmarking suites that Native Image was capable of compiling. A subset of benchmarks from Renaissance suite (MNEMONICS, PAR-MNEMONICS, and SCRABBLE) on which we performed the analysis conduct the data manipulation using Java 8 Streams. The RX-SCRABBLE benchmark solves

Table 8. Benchmark Running Time

Benchmark	Default	PGO	PGO-AOT-Inline	GraalVM-JIT	C2-JIT
h2	13,348 ms	10,200 ms	10,496 ms	6,526 ms	6,464 ms
fj-kmeans	4,709 ms	4,493 ms	4,456 ms	1,824 ms	1,919 ms
mnemonics	9,065 ms	5,812 ms	4,215 ms	2,723 ms	6,048 ms
par-mnemonics	8,329 ms	5,401 ms	3,988 ms	2,295 ms	5,111 ms
philosophers	23,889 ms	23,056 ms	23,010 ms	16,320 ms	5,809 ms
reactors	35,213 ms	31,133 ms	31,104 ms	17,985 ms	20,914 ms
rx-scrabble	406 ms	335 ms	320 ms	291 ms	330 ms
scala-stm-bench7	3,812 ms	3,089 ms	1,814 ms	1,603 ms	1,480 ms
scrabble	87 ms	63 ms	49 ms	43 ms	144 ms
apparat	13,283 ms	8,133 ms	8,374 ms	5,526 ms	9,860 ms
kiama	400 ms	342 ms	313 ms	210 ms	287 ms
scalac	1,539 ms	1,206 ms	1,174 ms	1,009 ms	1,241 ms
scaladoc	1,312 ms	1,011 ms	946 ms	855 ms	1,180 ms
scalap	189 ms	159 ms	155 ms	107 ms	128 ms
scalariform	544 ms	463 ms	461 ms	345 ms	458 ms
tmt	17,875 ms	10,159 ms	10,108 ms	7,025 ms	10,970 ms

the same problem as does the SCRABBLE benchmark but relies on the *RxJava* framework to do so. The PHILOSOPHERS benchmark solves the dining philosophers concurrency problem using the ScalaSTM framework, which is also used for the SCALA-STM-BENCH7 workload. The REACTORS benchmark consists of a message-passing workload, while the FJ-KMEANS benchmark performs K-means algorithm using the Fork/Join framework. DaCapo benchmark H2 executes a set of database transactions. The APPARAT benchmark from the Scalabench suite optimizes files with specific extensions, and KIAMA consists of a language-processing workload. The SCALAC, SCALADOC, and SCALAP benchmarks represent the Scala compiler, the Scala documentation generator, and the decoder for the pickled classfile information, respectively. The SCALARIFORM benchmark is a code formatter for programs written in Scala, and TMT embodies a tool for the unlabeled-code analysis.

5.1 Comparison with Other VMs and Inlining Algorithms

In the main experiment, we compare the performance of the five configuration runs. Three of them represent the native-image runs (*default-ee*, *pgo-ee*, *pgo-aot-inline-ee*), while the remaining two configurations stand for the HotSpot runs (*graalvm-ee-jit*, *c2-jit*). The *default-ee* configuration does not include any PGO, *pgo-ee* uses the existing profile-guided optimizations of Native Image (described in Section 4.1), and *pgo-aot-inline-ee* uses our new profile-driven compilation-scheduling and inlining algorithm on top of the existing profile-guided optimizations of Native Image. In this experiment, we used the fixed values of the inlining parameters in all of the configurations. The fourth configuration *graalvm-ee-jit* represents HotSpot JVM that uses the the Graal Enterprise Edition compiler in JIT mode and the fifth configuration *c2-jit* assumes HotSpot JVM using the default JIT server (C2) compiler.

The results of all five configurations are shown in Table 8, such that each column represents the results of one configuration in the following order: *default-ee*, *pgo-ee*, *pgo-aot-inline-ee*, *graalvm-ee-jit*, and *c2-jit*. Compared to the *default-ee* configuration, the proposed algorithm brings improvements between 10%–55% in 14 benchmarks, and in two benchmarks, the improvements are less than 10%.

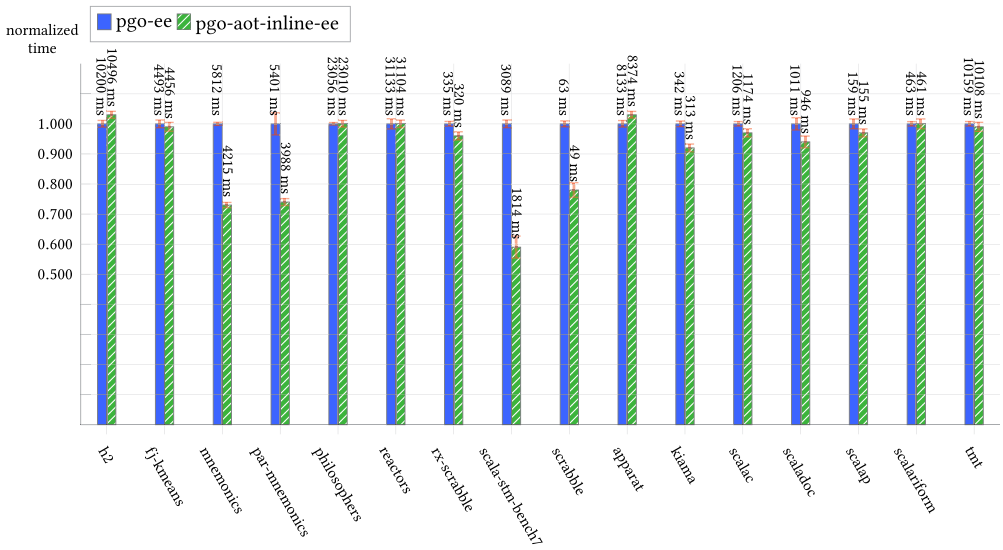


Fig. 6. Benchmark running time (lower is better).

The difference between *pgo-ee* and *pgo-aot-inline-ee* configurations is more relevant, since it directly shows impact of our new inliner compared to the previous use of the profiles. This is why we single out these two configurations and present their results also in Figure 6. The x-axis shows the benchmarks on which we conducted the main experiment. The plot contains two bars per benchmark, and each bar represents the results of a benchmark run with one of the two configurations. All performance results are normalized against the *pgo-ee* configuration and presented on the y-axis. The proposed algorithm has the best improvements on the benchmarks SCALA-STM-BENCH7, MNEMONICS, PAR-MNEMONICS, and SCRABBLE, improving their runtime by about 40%, 27%, 26%, and 22%, respectively. The KIAMA and SCALADOC benchmarks are improved in the range of 5%–10%, while the range of improvement for RX-SCRABBLE, SCALAC, and SCALAP benchmarks is between 2.5% and 5%. Five benchmarks show the improvement up to 1%, and in two benchmarks (H2 and APPARAT), we observed a slowdown of less than 3%.

5.2 Impact of the Inlining Parameters on Performance

To make sure that the performance of the configurations is fairly measured and compared, we ran the experiments with different parameters with an impact on the expansion and inlining during a method compilation and analyzed the results. In this article, we refer to the procedure of searching for the best inliner parameter values as *tuning*. Inliner tuning is an iterative search process whose goal is to find the combination of the parameters that leads to the best peak performance of the programs compiled with that inliner.

Tuning of the Inlining Parameters. The tuning process is conducted on parameters r, t_1, and t_2 from Equations (22) and (23). These parameters directly impact the amount of inlining in compilation units—*expansion inertia base-value* r drives the amount of call-tree exploration that the inliner performs, while *relative benefit coefficient* t_1 and *base target spending* t_2 drive the benefit threshold for deciding whether a method should be inlined, effectively limiting the budget that the inliner has available for inlining [97]. The inliner explores a larger part of the call tree for higher values of r and tends to make larger compilation units for smaller values of t_1 and larger values of t_2.

We tuned the parameters r, t_1, and t_2 separately, as follows: For the tuning of each parameter P, we first set the value of all the other parameters to their default values, which were previously

tuned for the existing Graal's inliner [97]. We then used a variant of the simplex algorithm [25, 61] to determine the range in which the optimal value of the currently-tuned parameter P is, keeping all the other parameters locked. Starting from the initial value P_0, we explored the values $P_0 \pm \epsilon \cdot 2^i$ at each step i until finding a range with an inflection point, that is, $[P_0 - \epsilon \cdot 2^j, P_0 + \epsilon \cdot 2^i]$ such that exists k for which $P_0 \pm \epsilon \cdot 2^k$ has a better fitness value than the boundaries of the range. For the fitness, we used the geometric mean across all benchmarks. Then, we divided that range into 10 to 15 equidistant steps and searched for the optimal fitness within these steps.

The fitness on individual benchmarks for the parameter r, while all other parameter values are locked, can be seen on the x-axis of Figure 7. We note that the process of tuning the parameters individually could be improved by multidimensional tuning of all the parameters, but we note that we had to build the native images of all the benchmarks for every datapoint (both the instrumentation and the optimized image), and simultaneous multi-parameter tuning would be much more expensive (it exceeded the amount of machine time available to us).

The process of tuning is performed using the same methodology and setup described in the beginning of Section 5. For each value of the parameter, we performed five separate measurements in different process instances, running the two main GraalVM Native Image configurations (*pgo-ee* and *pgo-aot-inline-ee*). Based on this tuning procedure, we chose the values $r = 550$, $t_1 = 0.0002$, and $t_2 = 300$ as defaults for our modified inliner.

Expansion-inertia Base Value. This parameter directly affects the amount of the call tree that gets explored before the inliner decides which parts of the explored call tree must be inlined and corresponds to the value r from Equation (22). We found that this parameter has the most significant impact on the performance of the inliner. In what follows, we show the effect of this parameter on the *pgo-ee* and the *pgo-aot-inline-ee* configurations.

Each plot in Figure 7 compares the performance of one of benchmarks for the two main run configurations across the interesting range of parameters. The x-axis represents the values of the expansion-inertia base value, while the y-axis shows the running time of the benchmark in the corresponding plot. The running time is expressed in milliseconds. We analyzed these plots to determine the best choice for the default parameter value. For the majority of the benchmarks, both configurations achieve the best performance when the parameter is between 400 and 550, with a few benchmarks achieving better performance between 1,000 and 1,300. The new inliner achieves better performance for all parameter values on 11 benchmarks; on REACTORS it achieves better performance for most parameter values; on APPARAT it achieves the same or better performance depending on the specific parameter values; on FJ-KMEANS and PHILOSOPHERS it achieves approximately the same performance across all parameter values; and is strictly worse on H2.

Figure 8 compares the best performance for some parameter value, chosen separately for each of the two configurations (i.e., the result of tuning each inliner on each benchmark separately). The different benchmarks are shown on the x-axis, and the normalized running time is shown on the y-axis (a lower value means better performance). The reference value 1.0 is the *pgo-ee* configuration, which does not use our new compilation and inlining policy. The *pgo-aot-inline-ee* configuration (the proposed inliner) achieves better peak performance on the 11 benchmarks and a similar performance on 3 benchmarks. These individually tuned results are relevant for AOT compilation, since offline compilation of different programs allows tuning the parameters for each program independently (this is less applicable to online JIT compilations).

5.3 Impact on the Compiled-code Size

We next demonstrate that the previously shown performance improvements incur an acceptable increase in compiled-code size. The goal is to validate that increasing the budget only for the hot compilation units does not significantly impact the compiled-code size, as argued in Section 3.

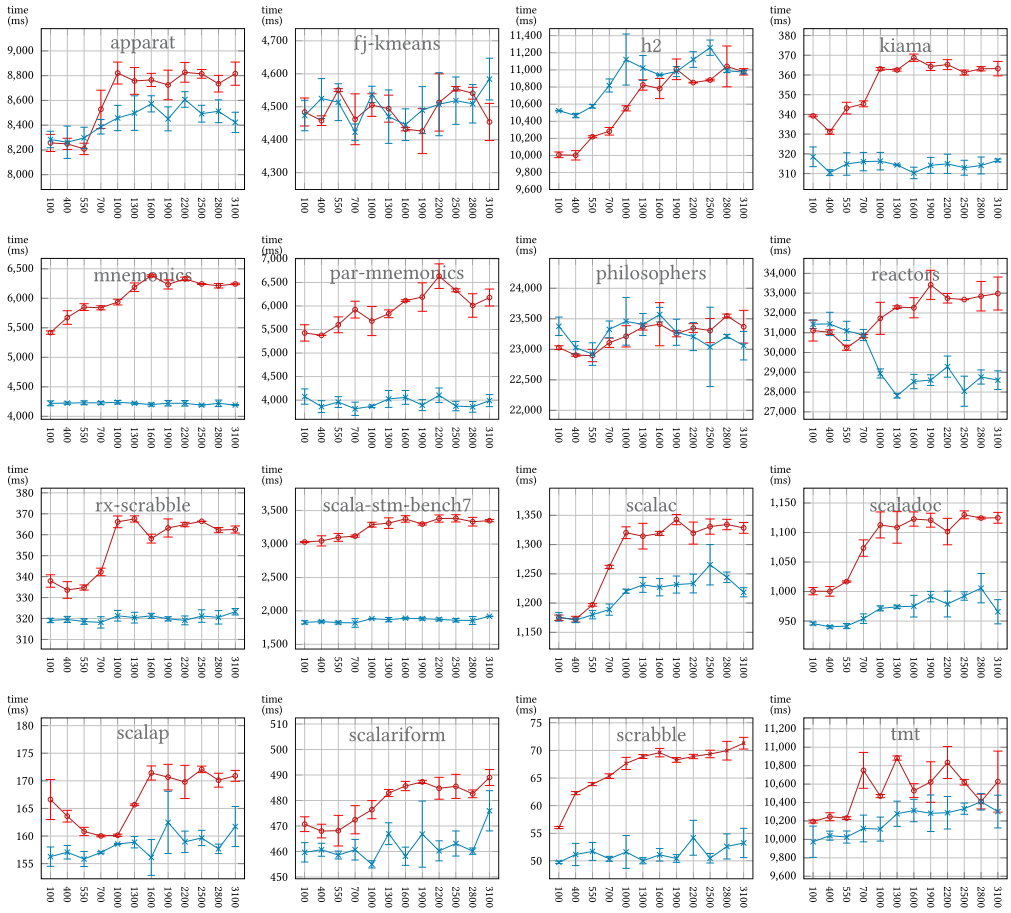


Fig. 7. Tuning results for the *expansion-inertia base value* parameter (*pgo-aot-inline-ee* (×) vs. *pgo-ee* (○))(lower is better).

Figure 9 compares the compiled-code size with the *pgo-aot-inline-ee configuration* against the compiled-code size with the *pgo-ee* configuration. We also compare the proposed algorithm against the approach of globally increasing the inlining budget for all the compilation units (i.e., not just the hot ones), which is represented by the *pgo-enhanced-inlining-budget-ee* configuration.

The x-axis shows the benchmarks, and the y-axis shows the compiled-code size. Each benchmark is associated with one bar for each configuration. The results are normalized against the *pgo-ee* configuration, which has the least amount of inlining. With our new inliner, the compiled-code size is increased in the range from 0.8% on the PAR-MNEMONICS benchmark, up to 9% on RX-SCRABBLE, compared to the *pgo-ee*. However, for 10 out of the 16 benchmarks, the size is increased only up to 2.5%. When increasing the inlining budget globally, the size-increase goes up to 2.5×. The smallest increase for *pgo-enhanced-inlining-budget* is around 9% for PAR-MNEMONICS, for which we witnessed the lowest size-increase for *pgo-aot-inline-ee* as well. However, the average size-increase for *pgo-enhanced-inlining-budget* configuration is approximately 2×, which we consider large. We conclude that the size increase of 0.8%–9% in the case of the *pgo-aot-inline-ee* configuration (the proposed algorithm) is not significant and is acceptable in practice.

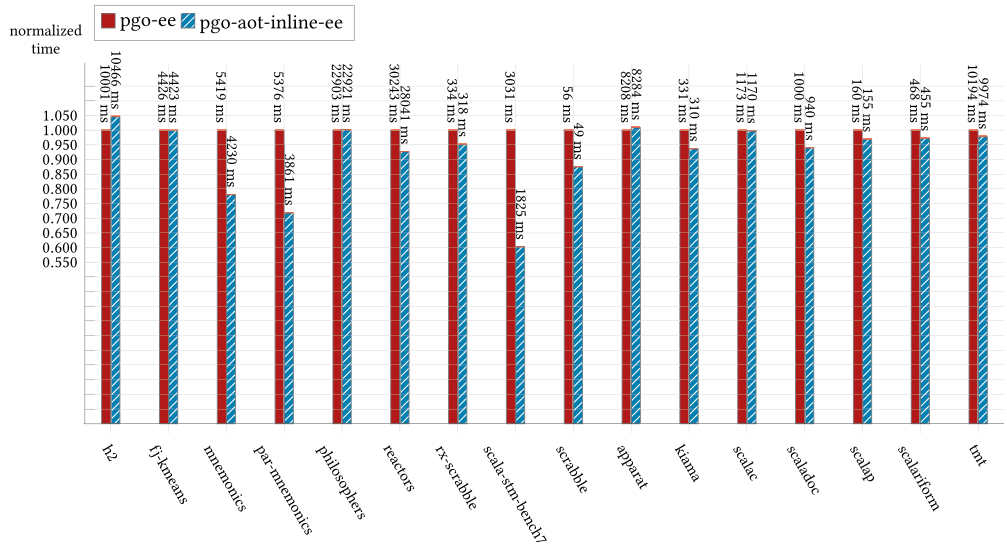


Fig. 8. Running time for per-benchmark-optimal *expansion-inertia base values* (lower is better).

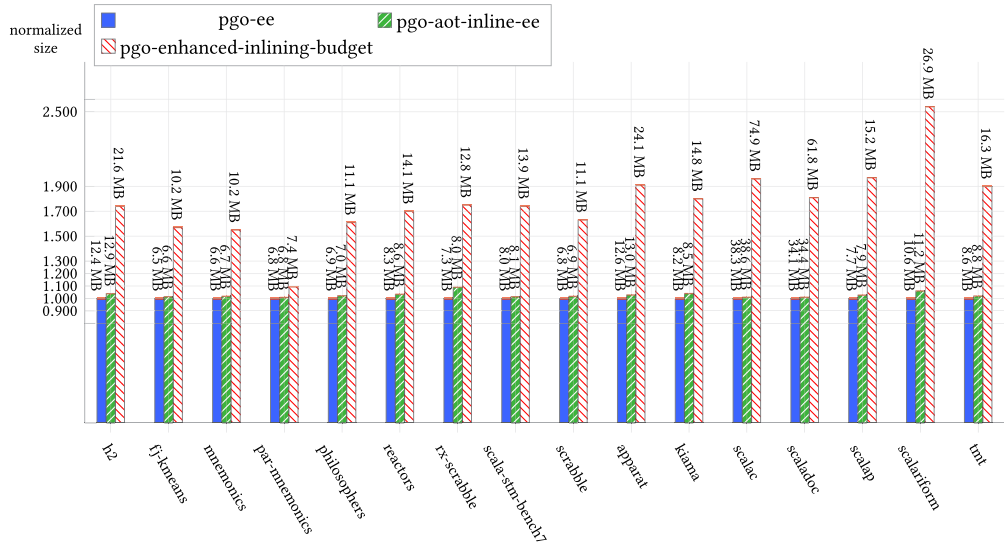


Fig. 9. Compiled-code size (lower is better).

5.4 Impact on the Compile Time

In this section, we present the impact of our algorithm on the compile time. Even though the compilation is performed ahead-of-time, and it is not crucial to keep the compilation overhead low, we demonstrate that the overhead is reasonable. Figure 10 compares the compilation time for *pgo-aot-inline-ee* configuration against the compilation time for *pgo-ee* configuration. The x-axis contains the benchmarks, and the y-axis shows the normalized compilation time. Each bar represents a compilation time of a benchmark run with a specific configuration. The results are normalized against the *pgo-ee* configuration. Our inlining algorithm increases the time of the compilation up

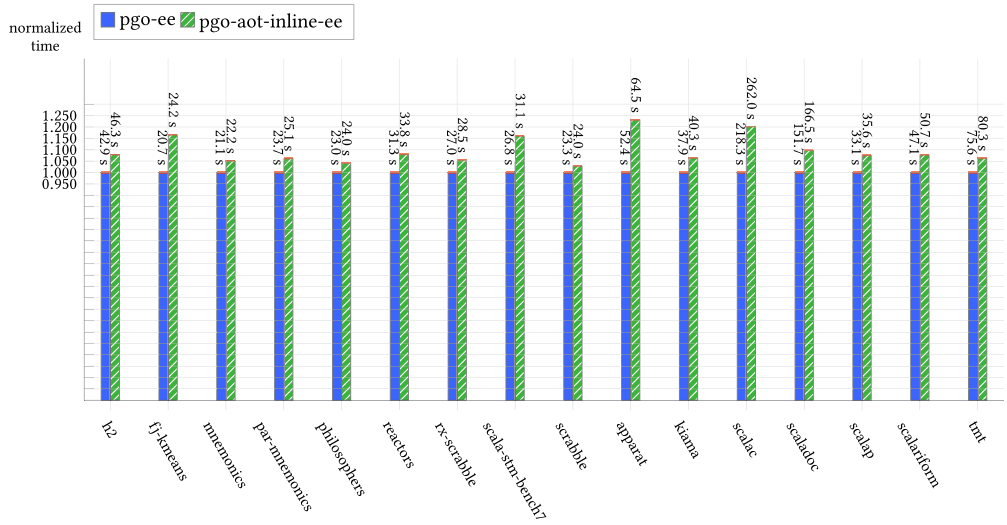


Fig. 10. Compilation time (lower is better).

to 23%, as observed on the example of the APPARAT benchmark. For 12 out of the 16 benchmarks, compile time is increased in range from 2.8% to 10%.

5.5 Impact of Context Length on Performance

One of the aims of the proposed algorithm is to utilize context-sensitive profiling information to improve inlining decisions. Context length is the number of locations in the calling-context of a profile entry. Longer contexts usually result in more precise profiling information. By varying the average context length of the input profiles, we show that there exists a dependency between the context length and the performance of the new inliner. The context lengths were varied by limiting the inlining depth in the instrumentation image and simultaneously boosting the inlining budget. For each benchmark, the depth was varied until boosting the inlining budget no longer resulted in longer average context lengths (the limit on the average context length eventually occurs because virtual calls cannot be inlined in the instrumentation image).

Figure 11 shows the subset of benchmarks on which we observed a noticeable impact of varying the context lengths. Our performance evaluation showed less effect on the rest of the benchmarks. The plots contain the performance data for the average context-length in the range from 1 up to the longest average context length we provided from the partial contextual information (3–4, depending on the benchmark). The APPARAT and SCALA-STM-BENCH7 benchmarks have the largest variation: 35%–40% higher performance with average context length between 3 and 3.5, compared to using context-insensitive profiles. The SCRABBLE benchmark shows an improvement of approximately 18%, TMT 12%, SCALAP goes up to 8%, while on the remaining benchmarks in Figure 11, the improvement is 4%–5% when the more precise profiles are present.

5.6 Impact of the Inliner Heuristics on Performance

In the following experiments, we assessed the impact of the specific algorithm features on the benchmark performance. In each experiment, we present results on a subset of benchmarks that show the greatest impact of the specific component of the algorithm.

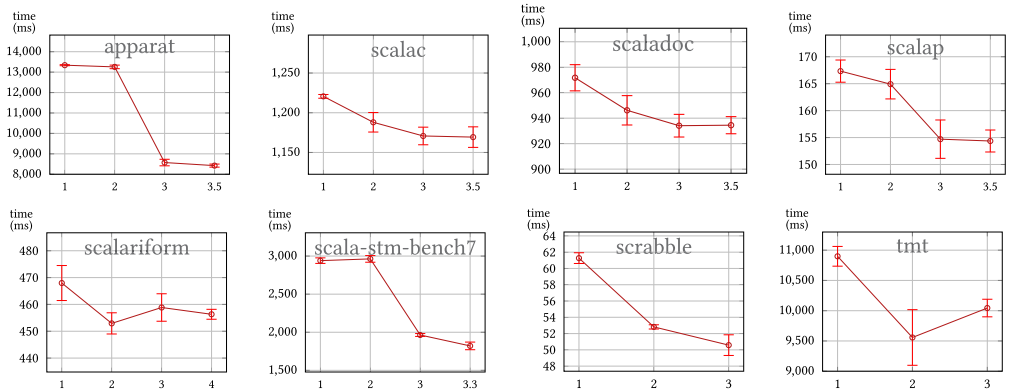


Fig. 11. Context-length impact on the benchmark running time (lower is better).

Table 9. Transitively Compiling Hot Callees as Hot Compilation Units

Benchmark	Hot Callees	No Hot Callees
scaladoc	938 ms	968 ms
scalap	157 ms	160 ms
mnemonics	4,208 ms	5,853 ms
par-mnemonics	4,003 ms	5,441 ms
reactors	30,582 ms	31,442 ms
scrabble	49 ms	63 ms
rx-scrabble	315 ms	323 ms

Transitive compilation of the hot callees with the increased inlining budget. After a method is compiled, its remaining non-inlined callees become the next compilation units. We refer to their compilation as *transitive*. These callees are compiled as either hot, with the increased compilation budget, or as cold, with the default budget—the specifics of deciding are explained in Section 3.6. In this experiment, we compared the proposed algorithm against a variant in which the IsHot policy from Equation (39) is replaced with simply IsHot = ⊥. In Table 9, column titled "Hot Callees" shows results of our unmodified algorithm, while the second column titled "No Hot Callees" shows performance when all callees are treated as cold. The improvement of the IsHot heuristic from Equation (39) is in the range of 22% to 28% in case of the `Stream`-based benchmarks, while for the other benchmarks, the improvement is up to 5%.

Trail expansion. The proposed algorithm iteratively expands the initial set of trails across the possible calling contexts. The details of this process are explained in Section 3.4, and the CallingContexts policy is defined by Equation (32). In this experiment, we disable the expansion by changing that policy to CallingContexts = ∅. Note that by excluding expansion from the algorithm, the set of hot compilation units contains only the methods that are the roots of the calling contexts of the initially chosen profile entries. Figure 12 compares the benchmark running time when the trail expansion is disabled (red curve) and the default setup with the trail expansion enabled (blue curve). The figure presents the initial hot-context threshold tuning process. Regardless of the threshold value, running time for benchmarks Apparat, Kiama, Scalariform, Scalac, Scaladoc, and Scrabble is strictly better when the expansion is active. Comparing the peak performance on both curves shows that the benchmarks profit up to 20% from this feature.

Size of the initial hot contexts set. The constant ψ from Equation (30) represents the proportion of total time that needs to be spent in a given profile entry to consider that entry hot for the initial set of profiles. In this section, we show that there is a range of optimal values of ψ when trail expansion is turned on; when the trail expansion is disabled, however, the optimal values of ψ are in a much narrower range, which varies across benchmarks.

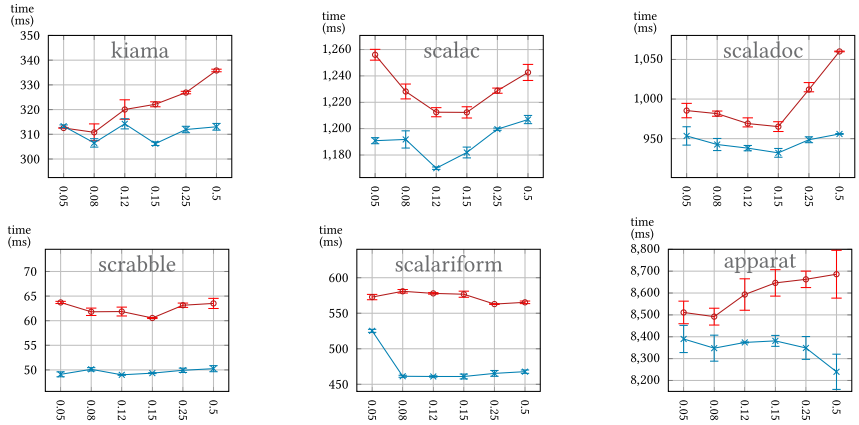


Fig. 12. Tuning the initial hot-context threshold (with expansion × vs. without expansion ○; lower is better).

Figure 12 contains a subset of the benchmarks that were particularly affected by the value ψ and shows how the different values of ψ affect performance. The x-axis contains the range of values of ψ, while the y-axis represents the running time. The greater threshold ψ implies less initial hot compilation units, i.e., less compilations with the increased inlining budget. Increasing ψ above a certain value leads to having no hot compilation units, which is the equivalent of the *pgo-ee* configuration. Based on the experiments with most benchmarks, we found that when the *trail expansion is turned off*, the optimal values of ψ lie on the range of 0.05% up to 0.5%—in this range, ψ varies the performance of the no-expansion variant by up to 10%. On most benchmarks, decreasing the amount of hot compilations below a certain threshold negatively affects the performance, as there are too many trails, and the best performance for the no-expansion variant is achieved for ψ between 0.12% and 0.15% (though, some of the benchmarks, as KIAMA, profit from the greater initial set of hot contexts). However, the range of optimal values of ψ for the variant that *does use trail expansion* is between 0.08% and 0.25% on most of the benchmarks, which indicates that trail expansion makes the algorithm less sensitive to the choice of ψ.

5.7 Performance Differences between GraalVM on HotSpot and Native Image

As shown in Figure 6, when the Graal compiler is used on the JVM (HotSpot or OpenJDK, which use just-in-time compilation), the performance is noticeably higher compared to GraalVM Native Image (which uses Graal in an ahead-of-time compilation mode). Despite the improvements of our inliner on GraalVM Native Image, there is still potential for achieving performance that is closer to that of GraalVM on the JVM. Aside from opportunities for better inlining decisions, we point out that there are other reasons for the observed performance difference between HotSpot and Native Image. In this section, we present the features and optimizations that are implemented differently in these two VMs (or not supported at all on Native Image). This is not a comprehensive overview of the differences in Graal's compilation on the JVM and on Native Image, but it shows some of the differences—in several cases, we analyze their performance implications by disabling specific features and optimizations on the JVM.

Snippets. Snippets are a mechanism in the GraalVM compiler used to express (in a restricted subset of Java) low-level implementations of high-level operations [111]. For example, `instanceof` (runtime type-check) against a class without subclasses can be expressed as a read of the object's header and a comparison against a constant. Similarly, a call to `System.arraycopy` (which copies

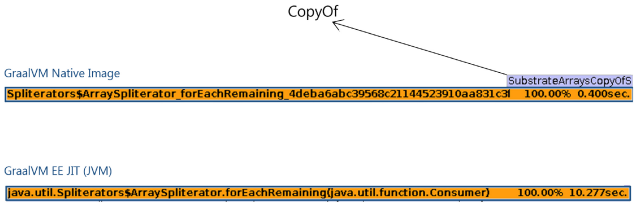


Fig. 13. Difference in the copyOf snippet implementation on native image and JVM (flame graph fragment).

elements between two arrays) can be expressed as a loop directly in the compilation unit that calls `arraycopy` and can be optimized when the arrays are not aliased and their types known. Most snippets are architecturally independent due to being expressed in a high-level programming language. However, they can contain platform-specific building blocks, in the form of special "intrinsic method calls" that map to, e.g., machine instructions, so snippet implementations vary across compiler configurations, underlying hardware platforms, and the VMs.

Consider the method `Arrays.copyOf` from the JDK, which duplicates a given array, and its intrinsifications on HotSpot (JVM) and on Native Image. Figure 13 contains call frames from the flame graphs of the MNEMONICS benchmark, both when Graal is used on HotSpot and within Native Image. Each frame in a call stack is a compilation unit, such that the callee units are placed on top of their caller units. The call stack in Figure 13 contains the compilation unit for the `ArraySpliterator.forEachRemaining` method from the Java `Stream` library. In the case of the Native Image flame graph, there is a separate frame for the `copyOf`, which is a specialized Native Image subroutine that does the array allocation and copying—the snippet merely calls the proper built-in method. On the JVM, there is no separate call—the snippet embeds the duplication logic directly into the IR of `forEachRemaining`. While for larger arrays, there is usually no observable performance difference between the two versions, for a lot of `copyOf` calls on smaller arrays, this results in a noticeable overhead.

Table 10. Impact of Array Copy Snippets in the Graal Compiler on the JVM

Benchmark	Default	No `arraycopy`	No `arraycopy` & `copyOf`
kiama	210 ms	278 ms	295 ms
mnemonics	2,723 ms	2,863 ms	5,470 ms
par-mnemonics	2,295 ms	3,604 ms	4,398 ms
scalap	107 ms	111 ms	113 ms
scrabble	43 ms	44 ms	47 ms
tmt	7,025 ms	7,905 ms	8,049 ms

In Table 10, we show the performance differences on six benchmarks on which we observed the highest impact of disabling Graal's snippets for `System.arraycopy` and `Arrays.copyOf` on the JVM. We realize that disabling these two snippets does not precisely correspond to the Native Image version, because the native C++ JVM implementation of `arraycopy` is different than the implementation of the `arraycopy` foreign call on Native Image. However, this experiment shows the magnitude of the impact that `arraycopy` and `copyOf` snippets have on these benchmarks.

Garbage Collection. The Java programming language uses **garbage collection (GC)** as the automatic memory-management technique. The JVM provides multiple GC implementations in each JDK version. We ran the benchmarks on GraalVM Enterprise Edition running on JDK 8, which by default uses the Parallel Garbage Collector [19]. Parallel GC freezes the application threads while performing the collection. However, unlike the Serial GC, which was the default in early JDK versions, Parallel GC uses multiple threads to perform garbage collection, which improves its throughput. The default garbage collector in Native Image VM is a serial garbage collector—it

pauses the application threads but uses a single collector thread [21]. Although the serial GC implementations on JDK 8 and on Native Image are different, it is useful to compare the performance of benchmarks running on JDK 8 with the Parallel GC and the Serial GC, as this gives a rough estimate of how much a different GC implementation would affect Native Image. Table 11 shows 4 benchmarks on which the Parallel GC considerably improves performance. The greatest impact on the benchmark's running time was observed for the `scrabble` benchmark—around 35%. Parallel GC speeds up most of the remaining 12 benchmarks by up to 5%.

Table 11. Comparisons of Garbage Collectors with the Graal Compiler on the JVM

Benchmark	Parallel GC	Serial GC
scrabble	43 ms	68 ms
scalac	1,009 ms	1,141 ms
reactors	17,985 ms	22,718 ms
kiama	210 ms	243 ms

Speculative Optimizations. JIT compilation on the JVM uses speculative optimizations that speculate about the characteristics of the values in the program to optimize the code better and improve performance. Each speculative code optimization is preceded by a computationally inexpensive check that confirms that the speculation is valid. If the check fails, then executing the optimized code would be incorrect, so a deoptimization is triggered—the execution is transferred to the interpreter, and the code is compiled again later, but with less speculations [65, 71]. AOT compilation on Native Image does not support deoptimization and therefore does no speculation. In this section, we show that speculations improve JVM performance on a subset of benchmarks.

Table 12. Impact of Speculative Optimizations in the Graal Compiler on the JVM

Benchmark	Speculations ON	Speculations OFF
apparat	5,526 ms	5,942 ms
h2	6,526 ms	6,647 ms
mnemonics	2,723 ms	2,849 ms
scalac	1,009 ms	1,053 ms
scalap	107 ms	116 ms
tmt	7,025 ms	7,963 ms

We identified three speculative optimizations in the Graal codebase for which we observed the highest impact: speculative guard motion [65] (which generalizes guard conditions to make them more loop-invariant [22]), optimistic aliasing analysis (which speculates that two pointers do not represent the same object), and speculative type-checking (which speculates that simpler type-check implementations can be tried first). These optimizations are also used on Native Image, but without the "speculation" part—they will never create a deoptimization point. To bring JVM closer to Native Image, we turned off the speculation in these optimizations (the optimizations are still enabled, but the `maySpeculate` calls return `false`, whereas in the default JVM setup `maySpeculate` returns `false` for a specific code location only after that location previously caused a deoptimization). Table 12 shows six of our benchmarks on which we observed a clear slowdown when disabling speculations: APPARAT, H2, MNEMONICS, SCALAC, SCALAP, and TMT. On these, slowdowns from disabling the aforementioned speculations range from 13% to 2%.

5.8 Profiling Impact on the Performance

To conclude the evaluation, we present the impact of the profiling on the compile time and the running time of the benchmarks. As we explained in Section 4.2, in GraalVM Native Image, profiling is realized in a separate compilation mode—by building an instrumentation image. The compilation pipeline in the instrumentation-image-build process contains a phase, which inserts the counter nodes in the IRs of each compilation unit. The profiles are obtained by running the instrumentation image. Compiling a program with the input profiles is a separate process, performed by building an optimization image. In the previous experiments in Section 5, we presented different aspects of the performance of the optimization image. While the algorithm presented in this article does

not change how the profiling itself is performed in GraalVM Native Image, we demonstrate how it affects the compile time and the running time of the instrumentation image on a set of benchmarks.

Table 13 demonstrates the impact of the profiling on the representative benchmarks from Renaissance, DaCapo, and Scalabench suites. To underline the effect of the instrumentation pass, we compare the time necessary to build and run the instrumentation image against the default image, which does not include any profile-guided optimizations. The performance of the benchmarks with the default configuration, in terms of running the default image, are presented and compared to the other configurations in Section 5.1. We observe that the instrumentation affects the compile time up to 30% for most of the benchmarks. However, on some benchmarks such as SCALAC, we observed the increase of 37%. The instrumentation may introduce the slowdown in the running times of the benchmarks up to 4×, which we recorded for REACTORS benchmark. Since the profiling is performed to a separate binary and run as a separate process, performance of the optimized image is not affected by the longer time needed to build and run an instrumentation image. We believe that improving the process of obtaining the profiles is possible, but it is out of scope of this article. Another benefit of having a separate process to collect the profiles is that, once they are stored to a file, the same file can be used for multiple optimization-image builds, i.e., compiling a program in an optimized configuration does not require compiling it and running in the instrumentation mode in case there is at least one profiling file for the same program from the previous builds.

Table 13. Profiling Impact on the Compile and Running Time

	Compile Time		Running Time	
Benchmark	**Default**	**Profiling**	**Default**	**Profiling**
h2	36.6 s	46.1 s	13.3 s	23.6 s
mnemonics	20.2 s	25.7 s	9.6 s	21.7 s
reactors	26.4 s	34.6 s	35.2 s	141.3 s
scalac	119.1 s	164.2 s	1.5 s	5.7 s
scalariform	38.4 s	46.3 s	0.5 s	1.1 s

6 DISCUSSION

The goal of this section is to discuss how the proposed algorithm in this article can be adapted to other compilers. For this purpose, we chose the two most commonly used environments—LLVM [7] and GCC [2]. To examine how our algorithm could be reused in LLVM and GCC, we focus on the profiling infrastructure, intermediate representation, detection of the frequently executed code, profile-guided optimizations, and the inlining optimization. To be clear, we did not reimplement or evaluate our algorithm in LLVM or GCC—the objective of this discussion is to demonstrate that our algorithm could be used in other compilers and to put our work into a broader context.

6.1 GCC

IR. GCC uses three major intermediate representations: GENERIC [4], GIMPLE [5], and RTL [6]. GENERIC is a tree-like language-independent representation of a program used by a compiler's front end. The midend of the compiler uses GIMPLE, a tree-like representation derived from GENERIC, in three-address form, as an additional restriction. GIMPLE tree is later on transformed into an SSA form, on which the majority of the optimizations is performed, including inlining. The last representation, RTL, is a low-level representation corresponding to a target architecture.

Hot-code detection. Many optimizations in GCC benefit from the information about the hotness of the code [79]. In GCC, hotness is determined on the basic-block level. A number of optimizations such as function inlining, block reordering, and loop unrolling, use the specific predicate, which indicates whether a basic block can be considered as hot, cold, or never executed. This predicate is used mostly to avoid aggressive optimizations on rarely executed blocks. When the profiling information is available, a predicate is set for each basic block based on the frequency of its execution

from the profiles, i.e., the hotness of the block. Predicates for hot, cold, and non-executed code are determined based on the thresholds that the block frequency must exceed and which can be tuned.
Profiling infrastructure. Similar to GraalVM Native Image, the profiles are collected and used in the following manner [24]: The program is first compiled with a flag that enables the profiling-by-instrumentation-phase. The output of running the resulting binary is a file containing the instrumentation profiles, which will be used as the input of the second compilation of the program. The instrumentation counters are placed in a program's CFG and produce the profiles containing the information about the number of function invocations, number of executions of basic blocks, number of executions of each edge in CFG (and therefore the corresponding basic blocks), from which the probabilities of taking a branch are derived [79].
Inlining optimization. The inliner relies on the bin pack algorithm. It uses numerous metrics such as the maximal size of a compilation unit, maximal size of the inlining candidates, the growth of large functions, and so on, to prioritize the inlining alternatives before reaching a limit [79]. When the profiling information is available, the inlining-candidate callees are prioritized according to a cost-benefit function that uses the hotness profiles and the growth of the function in which the inlining happens.

6.2 LLVM

IR. LLVM IR [11] is a strongly typed low-level, three-address representation in SSA form, which allows unstructured control flow and uses phi values when merging control paths. Transforming Graal IR to LLVM IR is a straightforward process, and all the basic compiler optimizations that work on the Graal IR can be implemented on the LLVM IR with a similar level of effort.
Hot-code detection. The inliner relies on several parameters to determine the threshold for deciding whether the code is hot or cold. Optimizations rely on the hot-code classification when the higher optimization levels are enabled. Hot/cold-splitting optimization uses the edge profiles to classify basic blocks of the current compilation unit into hot and cold. This is also useful for the optimizations such as function inlining and outlining, for which the LLVM implements outlining of the cold regions from the hot code [86].
Profiling infrastructure. Profiling is traditionally enabled through the instrumentation technique. Profiles that can be collected and exploited contain the information about the hotness of the invoked functions and edges, and receiver types and the number of their occurrences for the virtual invocations, similarly to the profiling information in GraalVM Native Image. It is possible to collect both the context-insensitive and callsite-aware profiles [12].
Inlining optimization. The inliner uses the hotness heuristic to filter out cold indirect calls and to focus on inlining the hot ones [8, 9]. Target methods corresponding to the most frequent receiver types from the virtual profiles are placed as direct calls if they exceed the set hotness thresholds [40]. The block-level hotness information is exposed to the inliner, which uses this information by defining the callsite hotness thresholds globally and relatively to the function entry [10]. Cost-benefit analysis takes into account the size of the callee-candidates and the function in which the inlining takes place. Heuristics are used to compute the cycle savings per call site and also to limit the amount of recursive inlining. Substantial amount of inlining budget is directed towards more aggressively and precisely performing the inlining optimization around the hot code.

6.3 Discussion

Above, we briefly described the relevant aspects of LLVM and GCC. In the following, we discuss how our algorithm could be potentially applied to those compilers.

Our hot-code-detection algorithm finds the hot inlining trails by expanding hot calling contexts in the profiles. In our implementation, input for the hot-code detection is a set of profiles that

determine the basic-block frequency and the call-target probability on the indirect calls. The profile data in GCC and LLVM environments contains the same kind of information. They support performing the instrumentation after the inlining phase, which as a consequence has that the calling contexts from the profiles are partially context-sensitive [27]. Our proposed hot-code detection algorithm can be applied to GCC's and LLVM's profiling data to produce the inlining trails. The trails can then be used to determine which callees are in the hot calling contexts and adjust the hotness predicate for the corresponding basic blocks accordingly.

Both described environments use hot-code-region-detection within multiple optimizations. They enable compiling the hot code and optimizing it with modified budget. Hotness information can be used in inlining, as well as to drive the budget of other optimizations. Both compilers define specific threshold values based on which an execution is considered as frequent or not. The thresholds used in our algorithm can be tuned to align with those.

Improving the inlining heuristics using the information of the code hotness is a topic of interest in the LLVM and GCC communities [3]. Generally, the aim is to improve the interprocedural optimizations by exploiting the hotness information, which is where the breadcrumb trails may be of assistance because they connect multiple subroutines. The inlining heuristics in GraalVM, GCC, and LLVM use the cost-benefit analysis that takes into account numerous metrics such as the size and the growth of the compilation unit to calculate the cost. Since both GCC and LLVM emphasize benefits of using the code hotness to prioritize the inlining of a hot call, our modification to the existing inliner, which assumes giving a greater inlining budget to such calls, can be easily incorporated. The inlining budget itself is determined through a different set of inliner parameters, defined by the heuristics of the inliner. Therefore, the inlining parameters would have to be tuned for the peak performance with the rest of the compiler's optimization infrastructure.

7 RELATED WORK

In this section, we present a survey of the related work on the profiling, and the usage of the profiling information to aid compilation, inlining heuristics, and other compiler optimizations. We first present an overview of the common profiling techniques and their evolution over time. We then review the prior work that focused on acquiring and applying partially contextual profiling data and on the approximation of fully context-sensitive profiling data using partial profiling information. Finally, we compare our compilation technique with alternative techniques and optimizations that rely on profiles, with a special emphasis on inlining algorithms. Additionally, this section contains an extension of the Graal intermediate representation explanation from Section 4.1.

Profiling strategies. One of the earliest uses of profiling was described by Knuth [83], who defined the *profile* as a set of execution counts collected during the runtime of the program. Over time, the notion of profiles was expanded to include any metric that describes the program behavior and that was collected during the execution of the program. The inception of profiling has raised the questions of how the profiles can be exploited to optimize the execution of computer programs, which information they should contain, and on how to decrease the cost of profile collection [102].

Broadly speaking, profiling can be classified as either instrumentation-based or sampling-based. So far, numerous authors have observed that the exhaustive instrumentation (i.e., instrumentation of all the code of the program) provides more precise data compared to the sampling-based profiling [56, 64]. Some versions of the Netbeans IDE [17] and the Eclipse Test and Performance Tool Platform [14] come with the instrumentation-based profiling tools. However, the instrumentation is usually not suitable for online collection of the profiles, because it imposes a high overhead on the performance [116]. According to Arnold et al. [36], the instrumentation can degrade performance by between 30% and 1000%. The GraalVM Native Image, in which we implemented

our algorithm, assumes the offline profile collection, which is why it is acceptable to use the instrumentation to gather more accurate execution counts of the profiled events.

Some GCC compiler [56] and LLVM [94] extensions can consume profiles from external tools to assist compiler optimizations. Most compiler optimizations benefit from the profiling data: inlining [54], block ordering, path-duplication [88], register allocation [70], and many others [98]. The ahead-of-time Scala Native compiler [110] supports instrumentation-based profiling and uses profiles to guide optimizations such as devirtualization, method duplication, and block placement.

Profiler Classification. Another method of classification is based on whether a profiler uses an online or an offline profiling scheme [117]. Offline profiling assumes collecting profiles in the separate program runs after which there is a compilation process using them. AOT compilers usually exploit offline profilers. The GCC compiler has a mode for producing instrumented binaries for profile collection and can use these profiles when creating the optimized binary [20]. Online profiling is performed during the same program run and is mostly exploited by JIT compilers. Most VMs use JIT compilation, do profiling in the first stage [15, 23, 50, 80, 122], and in some cases also profile their JIT-compiled code [49]. Flückiger et al. [72] proposed two-tier JIT compilation for R language that uses instrumentation in the first tier to optimize default code and a sampler on the optimized code to trigger reoptimization in the second tier with more type-specific code. Unlike `gprof`, which is primarily used to profile programs that are AOT compiled, most of the profilers, such as *Valgrind* [28], *Java Mission Control* [26], *JProfiler*, *YourKit*, `hprof`, and `perf`, are not tightly coupled with either AOT or JIT compiled code. In other words, they can profile both. Jikes RVM [32, 34] uses an instrumentation-sampling framework to perform profile-guided optimizations such as method splitting [52], method inlining, loop unrolling, and code motion.

Partial profiling information. While the exhaustive instrumentation enables the highest accuracy, it usually benefits only applications that need very precise information, such as intrusion detection or debugging [48]. Several authors investigated the question of whether incomplete profiling information can be sufficiently precise for common compiler optimizations and how profiles can be approximated [31, 89, 90]. It was observed that keeping the full calling contexts starting from the root of the program does not have a substantial impact on the profile-guided optimizations, and furthermore, storing fully calling-context-sensitive profiles increases the footprint of the data structures that stores the profiles. Various authors therefore argued for having length-bounded calling contexts [31, 101].Ausiello et al. [37] collected the profiles with the calling contexts of length at most k, using a data structure called ***k-calling-context forest (kCCF)***, which consumes less space. The authors concluded experimentally that shorter calling contexts of k between 10 and 20 are typically sufficient to detect the most important control-flow paths. While the Native Image PGO supports efficient generation of partial calling contexts, it also dictates the value of k. As shown in Section 5.5, the average value of k is between 3 and 5 (depending on the workload), even though individual calling contexts can be several times longer.

Serrano and Zhuang [106] obtain partial call traces using hardware-level tracing and use them to reconstruct calling context information. In their technique, partial traces are merged if they contain a significant overlap. The result of the merge is a ***partial calling-context tree*** **(PCCT)**, which is similar to our trail data structure from Listing 4.

Although Serrano and Zhuang solve a problem that is similar to ours (merging smaller PCCTs into larger PCCTs corresponds to our trail grafting), there are four important differences between our proposed technique and the technique due to Serrano and Zhuang:

(1) In our input profiles from Native Image PGO, *there is no overlap for the same position in the activation tree*. One node in the activation tree is always represented by exactly one profile entry. This is because Native Image PGO obtains profiles by instrumenting each compilation unit with unique counters, and each node in the activation tree is covered by exactly one

compilation unit. While randomly collected trace samples exhibit overlaps to some degree, in our profiling data, for an entry with the calling context $\ell_1, \ldots, \ell_m, \ldots, \ell_n$, there is no entry $\ell_m, \ldots, \ell_n, \ldots, \ell_p$ that could be an overlap in the activation tree.

(2) Due to the differences in the inputs, the design of the two algorithms is different. While the technique due to Serrano and Zheng finds overlaps between the profiles to merge the partial call trees together, the profiles in our technique have no overlaps, so their technique cannot be applied directly to our input profiles. Instead, our technique extends the partial contexts by determining the possible callers. This makes the reconstruction problem considerably harder, in our view.

(3) The length of the traces that are available to us are typically shorter than those in Serrano and Zhuang's technique. As we show in Section 5.5, the average context length is between 3 and 4 call frames, while the length of the contexts in Serrano and Zhuang's technique depends on the hardware-sampling window size and therefore can reach much longer paths (their paper mentions path sizes of up to 32). The reason is that the Native Image PGO cuts a partial context at a virtual call (which the inliner cannot inline in the instrumentation image, because this image does not have a profile input).
There is a considerable consequence of having longer calling contexts—Serrano and Zhuang report that for partial call trails of length 16 and above, over 80% of reconstructed calling contexts have single callers, and of length 32 and above, that percentage is even greater and exceeds 90%. However, they observed that the percentage is significantly smaller for shorter call trails. Our technique is tailored to reconstructing call trees from shorter contexts, and as such has to deal with multiple callers more often (which is challenging).

(4) Finally, because we perform the expansions more aggressively, we dedicate a large part of the work to approximating the frequencies of the individual nodes in the reconstructed trail (i.e., partial call tree) to determine which of the possible callers more signifficantly contribute to the hotness of the code and to aid the inliner with more precise frequency information later. In particular, we introduce the attenuation factors (Equation (33)) each time we extend the trail upwards to address the fact that the hotness of a trail is the sum of the hotnesses across each caller.

Profiles in hot-code detection and inlining. Profiling is widely used to assist various compiler optimizations [20, 29, 34, 56, 72, 103]. While it is difficult to present a complete account of all the optimizations that benefit from profiling information, path duplication [88], inlining [33, 54], polymorphic inlining [76, 78], speculative code motion [65], function outlining [123], register allocation [69], and profile-driven code motion [57] are only some of the notable examples. In this section, we focus primarily on hot-code detection and on using profiles to guide inlining decisions.

Mytkowicz et al. [93] used hot-method detection as a criteria for evaluation Java sampling profilers. This metric is important, since incorrect hot-method identification results in spending the optimization budget in code that is not frequently executed. The importance of hot-code detection was emphasized by Kistler et al.'s work [82] on continuous re-optimization of the hottest code. To identify the hot code and inline the relevant callsites, Krintz [85] extended JikesRVM [29] with both online and offline profile collection and used the combined profiles to annotate the hot methods, and the callsites to inline. In that work, the estimation of method hotness is calculated based on the method invocation count, and the inlining of hot callsites is based on context-insensitive decisions.

A number of inlining heuristics do cost-benefit analyses to decide whether to replace a callsite with the body of the target subroutine [8, 33, 51, 53, 54, 59, 97, 104, 105, 124]. While some inliners mainly use static information such as the size of the methods [96], bypassing profiles usually leads to poor performance. In most compilers, inliners rely on frequency and receiver-type profiles.

Arnold et al. [33] compared the performance of various inlining techniques and remarked that when the only information at disposal is static, all the callsites of the same method are either inlined or not, regardless of the callsite frequency. The other approaches they evaluated used profiling information to compute the inlining benefit. Their results show that more context-sensitive information allows an inliner to make better decisions and achieve better peak performance. Scheifler's global inlining algorithm [104] prioritizes the inlining of the most frequently executed methods while ensuring that the global code-size constraints are not violated—the (context-insensitive) invocation frequency is obtained from the profiling runs of the program.

A compilation technique by Suganuma et al. [115] relies on a hybrid form of profiling to detect the most important parts of the program and to optimize them more aggressively. They use the simplest heuristics based on the static information to optimize the smallest methods, while the expensive optimizations such as inlining are conducted selectively on hot program sections. Their sampling technique determines the hot methods that should be compiled, and these methods are subsequently instrumented to obtain more precise profile information about the hot code. The profiles that they gather are context-insensitive, which impacts their precision.

Inlining is hindered by indirect calls (i.e., virtual dispatches), which are common in object-oriented and functional languages—since the call target is not known, inlining cannot be done accurately. This problem is ameliorated by profiling the receiver types at individual indirect callsites. As described by Grove et al. [76] and Hölzle et al. [78], polymorphic inlining can significantly improve program performance. However, polymorphic inlining is only as good as is the quality of the profiles, and on larger workloads, context-insensitive profiles have a tendency to get more and more polluted. The experiments done by Grove et al. [76] indicate that context-sensitive receiver-type profiles significantly improve the performance on the larger workloads. In our solution, trail expansion is performed directly from the profiles and, as such, works correctly. However, trail-grafting operation can attach more callees to a trail that would normally be callable for that callsite, but we rely on the inliner to prune those extra callees by considering only callees that were seen according to the receiver-type profile.

Instead of designing the algorithm with fixed inlining parameter values for all workloads, Cooper et al. [60] proposed having program-specific inlining decisions and heuristics. They described a system that adaptively tunes the inlining parameters for a specific program by efficiently searching the parameter space and have demonstrated that having program-specific heuristics and parameters results in best overall inliner performance.

Møller and Veileborg [92] present a static analysis algorithm for optimizing JDK 8 Streams library, whose goal is to transform functional `Stream` expressions to manually written loop equivalents. Their system works by identifying the `Stream` operation and then performing ahead-of-time inlining of its functions. It exploits the observation that, usually, the entire construction and execution of a `Stream` pipeline is located in a single compilation unit, then inlines all of the methods into that compilation unit and performs escape analyses and read-write eliminations to simplify the `Stream` operation into a simple loop. Møller and Veileborg demonstrated on simple `Stream` programs that their algorithm produces the correct result. In our work, we strived to provide library-agnostic inlining improvements, and we based the inlining decisions on partially context-sensitive profiles—we did not employ any library-specific knowledge or static analyses. We believe that inlining and compilation scheduling can be further improved by integrating Møller and Veileborg's analysis into trail-expansion heuristics, but we leave that to future work.

8 CONCLUSION

We presented a new algorithm that modifies the compilation schedule and the inlining decisions of an ahead-of-time optimizing compiler with the aim of improving program performance without

significantly increasing the size of the generated machine code. The algorithm utilizes the partially context-sensitive profiles, collected during offline profiling runs, to reconstruct call-tree fragments, called *trails*, which lead to hot parts of the program, and it uses these trails to influence the inlining behavior. We formally presented the algorithm and then described the implementation in GraalVM Native Image, a state-of-the-art ahead-of-time compiler for the Java programming language.

Evaluation on standard benchmark suites shows performance improvements in the range of 2.5%–40%, with only 2 out of 16 benchmarks from the evaluation showing a small slowdown of less than 3%. The size of the generated code is increased by 0.8%–9%, and in 10 out of 16 benchmarks, the increase is less than 2.5%. We conducted several additional experiments that show the breakdown of performance impacts and the impact of the various parameters on performance.

The algorithm can be implemented in most other optimizing compilers and is generally useful for other programming languages and runtime environments. Our formalization separates the algorithm from the policies that it can be tuned with, which simplifies the exploration of better policies and heuristics. For example, we believe that augmenting the policies in the algorithm with interprocedural analyses can result in additional performance improvements, but we leave these investigations to future work.

APPENDICES

A COMPILATION UNIT EXAMPLE

Having demonstrated the algorithm on a simple example, we illustrate its execution on the MNEMONICS benchmark from the Renaissance benchmarking suite [99]. Our algorithm identified 12 hot methods for compilation, one of them being `org.renaissance.jdk.streams.MnemonicsCoderWithStream.encode`. In this section, we show the compilation of the `encode` method and how the trail is applied to the inlining tree.

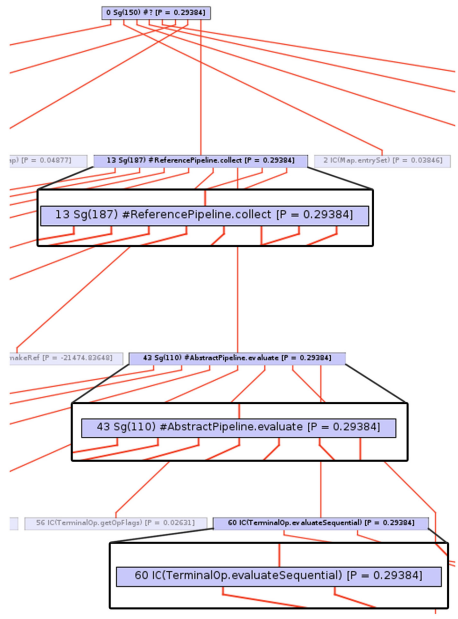


Fig. 14. Inlining tree fragment for the encode method from the MNEMONICS benchmark.

Figure 14 shows a fragment of the inlining tree of the `encode` method, created by the Graal compiler's inlining algorithm, and visualized with the **Ideal Graph Visualizer (IGV)** tool [121]. Several nodes were zoomed in and highlighted to make their captions more readable. The caption of each node consists of the node's unique ID, followed by a string `Sg` and the node-count of the respective method, name of the method, and the exploration priority of the inlining tree node (in square brackets). The root node corresponds to `encode`, and its direct child nodes correspond to a set of callees of `encode`. One such callee method is `ReferencePipeline.collect`, which has children of its own, for example, the method `AbstractPipeline.evaluate` from the figure. Further down, one child of the `evaluate` method is the node for the `TerminalOp.evaluateSequential` method. These nodes represent the hottest parts of the trail and lead to the method that contains the main loop of the JDK `Stream` operation.

We next consider the trail that corresponds to the hot compilation unit `encode`. Figure 15 contains the nodes of the trail that directly correspond to the highlighted inlining tree nodes

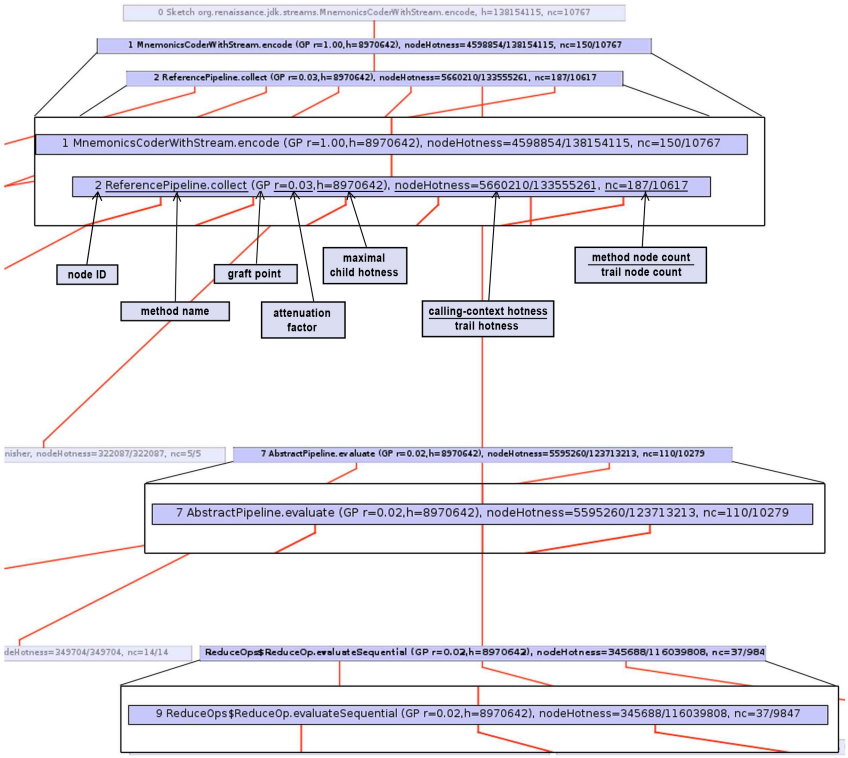


Fig. 15. Part of a trail for encode method from MNEMONICS benchmark.

in Figure 14. The highlighted nodes in Figure 15 show some of the relevant metrics such as the attenuation factor a_c, breadcrumb hotness κ, total subtree hotness χ, and the size ν expressed as the IR node count. The trail contains all the highlighted nodes from the inlining tree, in the same caller-callee order—exploration priority is boosted for all the inlining-tree nodes that can be mapped to the trail nodes.

In Figure 14, the exploration priority of these "on-trail" nodes is $P = 0.29384$, which is noticeably higher than the surrounding nodes, whose priority is in the 0.026 to 0.049 range. This larger value is a consequence of the modified expansion priority φ from Equation (21)—it forces the inliner to prioritize the exploration along the corresponding call chain and to explore the tree deeper around this region. Importantly, the exploration of other parts of the inlining tree still takes place, since the *penalty* function from Equation (20) reduces the exploration of subtrees that become too large.

B CASE STUDY: INLINING SCHEDULE IN THE MNEMONICS BENCHMARK

In this section, we analyze the impact of the proposed algorithm on the MNEMONICS benchmark. The aim is to demonstrate how our algorithm works on a *real-life* example from the evaluation benchmark. Figures 16 and 17 contain the most important parts of the flame graph for the AOT-compiled MNEMONICS benchmark, with and without the proposed inlining algorithm, respectively.

We use this example to illustrate the effect of two specific features of the proposed inliner: (1) We identify the compilation units that are hot and boost their compilation budget, and (2) we change the compilation order, which leads to a different set of compilation units. In both cases, we show how these features affect the benchmark by highlighting the relevant frames in the call stacks. To see the effect of increasing the inlining budget in hot compilation units, consider the call stack

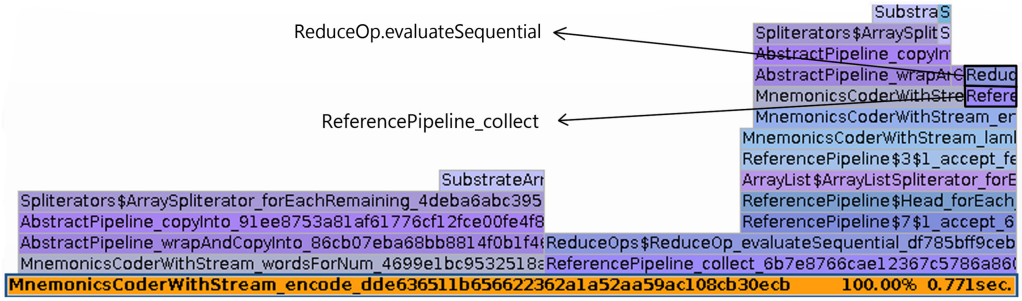


Fig. 16. Part of a flame graph for MNEMONICS benchmark with the proposed inlining algorithm.

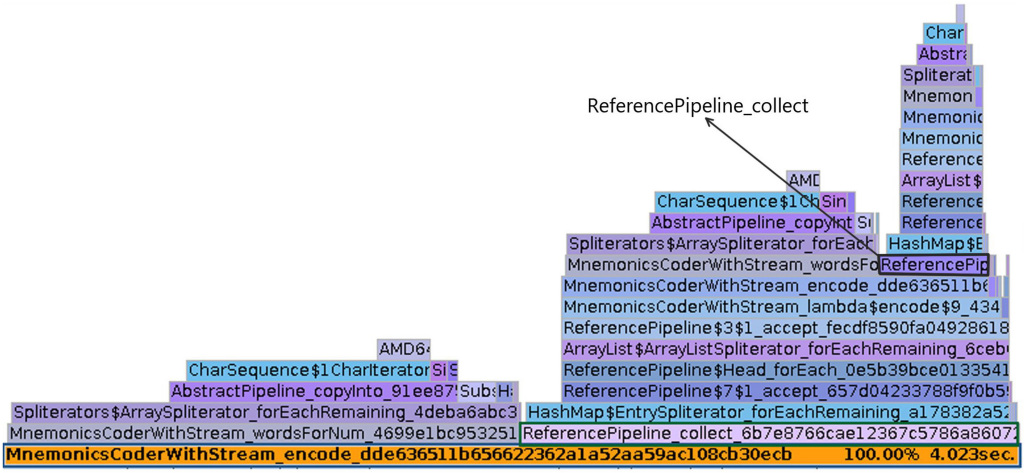


Fig. 17. Part of a flame graph for MNEMONICS benchmark with the original native image inliner.

on the left in Figure 16, in which the method `ArraySpliterator.forEachRemaining` is a hot compilation unit. The new inlining algorithm performs more aggressive inlining in this compilation unit—all the callees are inlined into `forEachRemaining`, except for the `arraycopy` snippet, as explained in Section 5.7. However, without the proposed algorithm, several callee methods, i.e., `AbstractPipeline.copyInto` (as well as their callees) remain separate compilation units in the flame graph, as shown in Figure 17 in the call stack on the left.

To see the effect of modifying the compilation order, note that the set of compilation units differs between the flame graphs in Figures 16 and 17. In the version with the proposed inlining and compilation-scheduling algorithm (Figure 16), the `ReferencePipeline.collect` compilation unit in the right call stack has a single remaining callee—method `ReduceOp.evaluateSequential`. In the standard version, this `evaluateSequential` is not a standalone compilation unit and is instead inlined in `ReferencePipeline.collect` method, which then runs out of compilation budget before inlining `HashMap$EntrySpliterator.forEachRemaining`.

C INLINER CHALLENGES FROM A USER PERSPECTIVE

In this section, we give several classes of code patterns that pose particular challenges to inlining algorithms. Our emphasis is on object-oriented and functional programming languages. Our goal is to illustrate some of the inlining problems from a programmer's perspective and provide possible mitigations that a programmer can apply (when the inlining algorithm is not sufficiently

sophisticated to apply them automatically). The suggested mitigations are not exhaustive, and the examples are not comprehensive—they are based on our own experience and performance analyses of individual programs.

Frequency profiles are polluted. Since inlining decisions are typically heavily guided by execution frequency profiles, a common problem is that the frequency information of a callsite is imprecise. Consider a frequency profile on a control-flow construct such as a loop. If the subroutine that the loop belongs to is called from many different callsites, then the frequency of the loop may vary for the different callers, but a context-insensitive profile will only report the average frequency. In such cases, we consider the frequency profile polluted. If the profile pollution results in an underestimate of the call frequency, then a performance critical call might not get inlined.

```
def sort(xs: Array[Int], lo: Int, hi: Int) {
  if (lo >= hi || lo < 0) return
  val p = part(xs, lo, hi)
  sort(xs, lo, p - 1)
  sort(xs, p + 1, hi)
}
def part(xs: Array[Int], lo: Int, hi: Int) = {
  var pIndex = lo - 1        polluted frequency profile
  var j = lo
  while (j <= hi - 1) {
    if (xs(j) <= xs(hi)) {   affected inlining decision
      pIndex += 1
      swap(xs, pIndex, j)
    }
    j += 1
  }
  swap(xs, pIndex + 1, hi)
  return pIndex + 1
}
```

Listing 6. Quicksort Algorithm.

Example: Listing 6 shows an implementation of the quicksort algorithm in the Scala programming language. The implementation consists of two functions `sort` and `part`. The `part` function partitions the subsequence using the last element as the pivot and places elements smaller than the pivot to the left of the subsequence and the larger elements to the right of the subsequence. The `sort` function partitions its subsequence by invoking `part` and then recursively invokes `sort` on the partitions.

The `while` loop in the `part` function is invoked many times recursively and for progressively shorter and shorter subsequences. Since most of the calls to `part` are near the leaves of the recursion tree, the averaged loop frequency is heavily biased towards those shorter loops. An inliner may therefore decide not to inline the `swap` call inside the loop body, which considerably changes the performance of the compiled code.

Mitigation: The programmer can artificially introduce context-sensitivity into the code by replicating the code parts, depending on the context in which they execute. In the concrete example in Listing 6, the programmer can repeat the `while` loop inside two branches, separated by a condition such as `hi - lo < THRESHOLD`. This ensures that the two copies of the loop get attributed with separate frequency profiles, which subsequently biases the inliner to inline the `swap` call at least in the hotter loop. Alternatively, the programmer can use metaprogramming facilities to declare the `part` function as a C macro, C++ template, or a Scala macro, and then instantiate its body twice, calling a different instantiation, depending on the difference between `hi` and `lo`.

Receiver-type profiles are too polluted. To be able to inline indirect calls, an inliner speculates on what the target subroutine is—it chooses several likely targets and emits code that checks what the target address (or receiver type) is and then inlines that implementation into the branch in which the check passes. By doing so, the inliner bets that this branch will be entered in most cases. The success of this technique depends on the quality of the call target profile—if the profile is too polluted, then the speculations done in the compiled code are rarely correct.

```
def foreach[U](f: A => U) {
  var i = 0
  while (i < table.length) {
    val entry = table(i)
    if (entry ne null) f(elem(entry))
    i += 1
  }
}
```

polluted receiver-type profile

Listing 7. Scala `HashSet#foreach`.

Example: Each collection in the Scala standard library has a `foreach` function that applies a user-specified function to each element of the collection. Listing 7 shows the `foreach` function of the `HashSet` collection, which traverses an array of entries and applies the user function `f` to each non-empty array entry. A call to function `f` is indirect, because the concrete implementation is unknown—an inliner must therefore rely on the receiver-type profile to devirtualize the call to `f`. Since `foreach` is one of the most commonly used collection operations in Scala, the receiver-type profile is as a rule polluted in all but the simplest programs.

Mitigation: The programmer must reduce the amount of profile pollution—one approach is to use metaprogramming when designing the library so common functions (such as the `foreach`) are implemented as Scala macros or C++ templates, which reinstantiate the code at each callsite. If the call to `foreach` is itself indirect, then users can manually insert a receiver-type-check for the types they expect to be common at a particular callsite and cast the receiver down to a concrete type before calling a function such as `foreach`. In the example in Listing 7, the user can also use an iterator and a `while` loop instead of the `foreach`, as that either eliminates the indirect call to the function `f` or creates a separate callsite for profiling, thereby reducing profile pollution.

```
def foldLeft[B](z: B)(op: (B, A) => B): B = {
  var result = z
  this.foreach(x => result = op(result, x))
  result
}
```

code-bloating nested indirect calls

Listing 8. Scala `TraversableOnce#foldLeft`.

Nested polymorphic inlining spends too much budget. The aforementioned profile-based devirtualization technique gets rid of indirect calls by inlining several likely targets and dispatching between them based on a receiver-type-check. The number of likely targets is small (usually up to three), but this still results in a code-bloat as the polymorphic inlining gets deeper. Since each level in the inlining tree speculates on more than a single implementation, each of which can contain nested indirect calls, the number of inlined combinations grows exponentially, even though only some paths in the inlining tree are actually executed at runtime. Most inliners are bounded by the size of the compilation unit, so the code-bloat forces them to stop before attempting to inline other parts of the call tree. Code-bloat also affects other size-driven optimizations.

Example: Listing 8 shows the implementation of the `foldLeft` operation for the `TraversableOnce` interface, which covers collections defined solely in terms of their `foreach` function. The `foldLeft` folds over the collection elements by starting from a zero element `z` and successively applying the operator `op` to the folded value and the collection element. The `foldLeft` calls `foreach`, passing it a function that calls `op` and updates the folded value `result`. Since both the `foreach` and the `op` call are indirect (the former because the type of `this` varies and the latter because the user-specified `op` parameter varies), several concrete targets are inlined for `foreach`, and then several concrete targets for `op` are recursively inlined for each of the speculated `foreach` targets.

Mitigation: If the language supports metaprogramming, then the programmer may again resort to macros to force instantiation. In the example in Listing 8, by defining `foldLeft` as a macro or a template makes the receiver of `foreach` and the `op` target visible, so the compiler can convert those indirect calls to direct calls. Even if those receivers are not visible, instantiating the `foldLeft` code at the callsite results in a new profiling point, which reduces pollution.

Low frequency callee code can optimize code of a high frequency callee. The bias that an inliner typically has towards high-frequency calls can spend the inlining budget before low-frequency calls are inlined. Some of the low-frequency calls may introduce additional type information that can enable subsequent optimizations, so not inlining them results in suboptimal performance.

Example: The Scala `aggregate` function, shown in Listing 9, is a collection operation that generalizes `foldLeft` and can have a parallel implementation. By default, it is implemented in terms of `foldLeft` in the `TraversableOnce` interface.

The signature of `aggregate` takes the zero-element `z` as a by-name argument, meaning that the argument is not immediately evaluated at the callsite—a function that represents the callsite-side expression is passed instead. In the `main` function, the `aggregate` function is used to return the sum of the lengths of all arguments. Here, the zero-element `0` is not immediately computed but is instead passed as a function that returns `0`. Consequently, even if the `foldLeft` from Listing 8 is inlined into the compilation of `aggregate` and simplifies to a simple `while` loop, the call to the function representing the by-name parameter `z` might not get inlined. Due to the use of erasure to implement genericity in Scala, if the by-name parameter `z` is not inlined, then it is returned as a boxed value of type `java.lang.Integer`. The boxed value subsequently causes boxing and unboxing in the `while` loop, which hurts performance at the place where most of the time is spent. Were the `z` call inlined, the compiler could more easily apply optimizations such as escape analysis.

Mitigation: The library designers should avoid by-name arguments (i.e., the `=>B` functions, which lazily evaluate the argument when referenced) and use by-value arguments instead. In the example in Listing 9, using `B` instead of `=>B` might have arguably been a better choice. More generally, if a low-frequency call is direct, then the user can force the inlining by using an `@inline` annotation when available. Without flow-sensitive analyses in the inliner, there is not much else that a user can do.

```
def aggregate[B](z: =>B)(
  seqop: (B, A) => B,
  combop: (B, B) => B
): B =
  foldLeft(z)(seqop)

def main(args: Array[String]) {
  aggregate(0)(_ + _.length, _ + _)
}
```

low-frequency optimization precondition

Listing 9. Scala `TraversableOnce#aggregate`.

Low frequency callee has hot code. Another common problem for call-frequency-based inliners is to inline calls that happen infrequently, but which themselves contain a lot of hot code (for example, a high-frequency loop). If the callee receives parameters from the caller that devirtualize callsites in the callee or enable escape analysis of the parameters allocated at the callsite, then not inlining such a low-frequency callee is typically detrimental to performance.

Example: The `foldLeft` function from Listing 8 manifests this problem—its `foreach` function is called only once from its body, but the `foreach` itself contains a high frequency loop whose body has several indirect calls that involve the parameters passed at its callsite.

Mitigation: For direct calls, the programmer may resort to metaprogramming or to the `@inline` compiler hint, when available. In the example in Listing 8, it is beneficial for the library designer to override `foldLeft` in heavily used collections so it contains a `while` loop instead of a `foreach` call. Another way to mitigate indirect calls is to speculate on the type of `this` in the programming language itself—the frontend compiler for a programming language that supports metaprogramming may speculatively instantiate all `foreach` implementations that are macros (to aid the inliner of the optimizing compiler of the runtime environment).

D ALGORITHM DETAILS

This section contains the details of the policies used in the algorithm to detect the most frequently executed code and to approximate longer partially context-sensitive profiles based on the input profiles. We also explain in detail the classification of the callees as either hot or cold. These algorithm components are introduced in Sections 3.4 and 3.6.

D.1 Hot-code-detection Policies Details

This section contains the definitions and the details of the key procedures for the hot-code detection from Algorithm 2, based on which we define a *hot-code-detection policy*.

Equation (30) formally describes creating a set of initial trails from the profile Π. We define a constant $\psi \in [0, 1]$ and mandate that the profile's contribution h_I to the overall hotness exceeds ψ.

$$\textsc{InitialTrails}(\Pi) \equiv \left\{ (N, E, \gamma) : (\langle \ell_1, \dots, \ell_n \rangle, h_I) \in \Pi \wedge \frac{h_I}{\sum\limits_{(L,h)\in\Pi} h} > \psi \right\}$$
$$\textit{where} \quad N = \{\eta_{\ell_1}, \dots, \eta_{\ell_1,\dots,\ell_n}\} \quad E = \{\eta_{\ell_1} \to \eta_{\ell_1,\ell_2}, \dots, \eta_{\ell_1,\dots,\ell_{n-1}} \to \eta_{\ell_1,\dots,\ell_n}\} \quad \gamma = \{\eta_{\ell_1}\} \tag{30}$$

$\textsc{DetectionDone}$ becomes true when all the trails from T end up in the final set F, as per Equation (31).

$$\textsc{DetectionDone}(T, F) \equiv T \setminus F = \varnothing \tag{31}$$

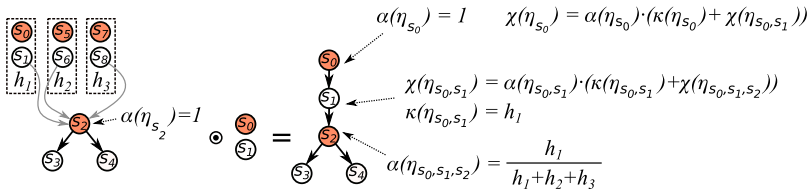
This termination condition therefore assumes (1) that F grows monotonically and (2) that T eventually stops growing.

$\textsc{CallingContexts}$ procedure. Represents a set of all the calling contexts from the profile entries, which can be used for extending a trail, and is defined in Equation (32). We identify the subset $\Pi|_c$ of profile entries in Π that refer to callsite executions specifically. For a specific trail τ, the set *callerProfiles* is determined as those entries from $\Pi|_c$ that end with a subroutine s_n and call $root(\tau)$ in the call-graph G (below, $\eta_1 \to \eta_2 \in G$ where $G = (N, E)$ is short for $\eta_1 \to \eta_2 \in E$).

$$\begin{aligned} &\textit{callerProfiles}((N, E, \gamma), \Pi, G) \equiv \{(\langle s_1, \dots, s_n \rangle, h) : (\langle s_1, \dots, s_n \rangle, h) \in \Pi|_c \wedge s_n \to root(N, E) \in G\} \\ &\textsc{CallingContexts}(\tau, \Pi, G) \equiv \{L : (L, h) \in \textit{callerProfiles}(\tau, \Pi, G)\} \end{aligned} \tag{32}$$

To estimate the portion of the trail $\tau's$ hotness for a particular extension $\tau \odot c$ the $\textsc{CallingContext}$ procedure computes an *attenuation factor* $a_c(\tau) \in [0, 1]$ of each context $c = \langle s_1, \dots, s_n \rangle$. Below, the numerator is the hotness of one particular calling context $c = \langle s_1, \dots, s_n \rangle$ that calls $root(\tau)$, and the denominator is the hotness sum of all the calling contexts L that call $root(\tau)$.

$$a_{s_1,\dots,s_n}(\tau) \equiv \frac{h_{s_1,\dots,s_n}}{\sum_{(L,h)\in \textit{callerProfiles}(\tau,\Pi,G)} h} \quad \textit{where} \quad (\langle s_1, \dots, s_n \rangle, h_c) \in \Pi \tag{33}$$



We will use the same example figure from Section 3.4 (presented above) to illustrate the definitions of the attenuation factor, trail hotness, and breadcrumb hotness.

Trail $s_3 \leftarrow s_2 \to s_4$ (on the left) has the calling contexts $s_0 \to s_1$, $s_5 \to s_6$, and $s_7 \to s_8$, with hotness h_1, h_2, and h_3, respectively. The attenuation factors of these different calling contexts are computed using the Equation (33). To compute the hotness of the trail, we need to sum together the hotness of the breadcrumbs but weigh them using the attenuation factors. We define three functions to serve this purpose: graft-point attenuation α, which records the attenuation factor when the trail gets extended; breadcrumb hotness κ, which records the hotness of the individual node; and the trail hotness χ. To illustrate this, we use the example from the preceding figure. In the figure, the root attenuation $\alpha(\eta_{s_2})$ of the trail $s_3 \leftarrow s_2 \to s_4$ is 1, but after the trail is extended, the attenuation of that node becomes $h_1/h_1+h_2+h_3$. Next, the hotness κ of an individual breadcrumb

η_{s_0,s_1} is inherited from the calling context hotness h_1. The hotness χ of the trail is computed recursively by summing the hotness of the children and the current node and multiplying it with the attenuation α.

Trail hotness. The *trail hotness* is a function $\chi : \mathbb{T}(P) \to \mathbb{R}_0^+$ that maps a trail to a non-negative real value. For its definition, we use the auxiliary functions α and κ.

Definition D.1. Graft-point attenuation of a trail τ is a function $\alpha_\tau : N \to [0,1]$ that maps each node to a real value between 0 and 1. For any trail $\tau = (N, E, \gamma)$, $\alpha_\tau(\eta) = 1$ for all the nodes that are not graft-points, i.e., $\eta \notin \gamma$. For graft-points $\eta \in \gamma$, the algorithm incrementally constructs α_τ when the trail τ is created. The rules are as follows:

- For all the trails that are initially created from some profile entry $(\langle \ell_1, \dots, \ell_n \rangle, h) \in \Pi$ (see Equation (30)), the algorithm sets $\alpha_\tau(\eta_{\ell_1}) = 1$. That is, attenuation is 1 on new trails.
- For every trail $\tau = (N, E, \gamma)$ that is the result of grafting $\xi \circledast_{\eta_{s_1,\dots,s_n}} \rho$ (see Equation (9)), where $\xi = (N_\xi, E_\xi, \gamma_\xi)$ and $\rho = (N_\rho, E_\rho, \gamma_\rho)$: If $\eta \in \gamma_\rho$, then $\alpha_\tau(\eta) = \alpha_\rho(\eta)$; otherwise, if $\eta_{s_n,d_2,\dots,d_m} \in \gamma_\xi$, then $\alpha_\tau(\eta_{s_1,\dots,s_n,d_2,\dots,d_m}) = \alpha_\xi(\eta_{s_n,d_2,\dots,d_m})$. That is, attenuation is inherited from the inputs.
- For every trail $\tau = (N, E, \gamma)$ that is the result of extension $\rho \odot c$, where $\rho = (N_\rho, E_\rho, \gamma_\rho)$ and $c = \langle \ell_1, \dots, \ell_n \rangle$, the algorithm computes an attenuation factor $a_c(\rho)$ for each context c (as defined in Equation (33)) when the context c is created in the CallingContexts procedure in line 6 of Algorithm 2. Then, for the root graft-point $\alpha_\tau(root(N, E)) = 1$; for the previous root $\alpha_\tau(\eta_{\ell_1,\dots,\ell_n,sub(root(N_\rho,E_\rho))}) = a_c$; and for all other $\eta_{s_1,\dots,s_m} \in \gamma_\rho$, $\alpha_\tau(\eta_{\ell_1,\dots,\ell_n,s_1,\dots,s_m}) = \alpha_\rho(\eta_{s_1,\dots,s_m})$ That is, the α_τ of the graft-point that was previously the root is set to the value a_c, and the attenuation of the remaining graft-points is kept unchanged.

Definition D.2. Breadcrumb hotness of a trail $\tau = (N, E, \gamma)$ is a function $\kappa_\tau : N \to \mathbb{N}_0$, which maps each node to its estimated count. The algorithm constructs κ_τ when τ is created, as follows:

- For all the trails that are created from a profile entry $(\langle \ell_1, \dots, \ell_n \rangle, h) \in \Pi$ (see Equation (30)), the hotness of the deepest node is $\kappa_\tau(\eta_{\ell_1,\dots,\ell_n}) = h$, and $\kappa_\tau(\eta) = 0$ for all other nodes.
- For every trail $\tau = (N, E, \gamma)$ that is obtained by grafting $\xi \circledast_{\eta_{s_1,\dots,s_n}} \rho$ (see Equation (9)), where $\xi = (N_\xi, E_\xi, \gamma_\xi)$ and $\rho = (N_\rho, E_\rho, \gamma_\rho)$: If $\eta_{s_n,c_2,\dots,c_m} \in N_\xi$ and $\eta_{s_1,\dots,s_n,c_2,\dots,c_m} \in N_\rho$, then $\kappa_\tau(\eta_{s_1,\dots,s_n,c_2,\dots,c_m}) = \kappa_\xi(\eta_{s_n,c_2,\dots,c_m}) + \kappa_\rho(\eta_{s_1,\dots,s_n,c_2,\dots,c_m})$; if $\eta_{s_n,c_2,\dots,c_m} \in N_\xi$ and $\eta_{s_1,\dots,s_n,c_2,\dots,c_m} \notin N_\rho$, then $\kappa_\tau(\eta_{s_1,\dots,s_n,c_2,\dots,c_m}) = \kappa_\xi(\eta_{s_n,c_2,\dots,c_m})$; otherwise, $\kappa_\tau(\eta) = \kappa_\rho(\eta)$. I.e., the hotness of the grafted nodes is added together where possible and inherited otherwise.
- For every trail $\tau = (N, E, \gamma)$ that is produced by the extension $\rho \odot c$, where $\rho = (N_\rho, E_\rho, \gamma_\rho)$ and $c = \langle \ell_1, \dots, \ell_n \rangle$: Nodes that originate from ρ inherit the hotness, i.e., $\kappa_\tau(\eta_{\ell_1,\dots,\ell_n,s_1,\dots,s_n}) = \kappa_\tau(\eta_{s_1,\dots,s_n})$. The newly created nodes $\kappa_\tau(\eta_{\ell_1,\dots,\ell_i})$ have their hotness set to 0.

Definition D.3. The trail hotness χ is then defined as the recursive hotness sum of all the breadcrumbs in the trail τ, where each subtree is weighted with the graft-point attenuation α_τ:

$$\begin{gathered}
\chi(\tau) \equiv \alpha_\tau(root(N,E)) \cdot \left(\kappa_\tau(root(N,E)) + \sum_{\tau_c \in subtrees(\tau)} \chi(\tau_c) \right) \quad where \quad \tau = (N, E, \gamma) \\
subtrees(\tau) = \{(N_{\eta_c}, E_{\eta_c}, \gamma_{\eta_c}) : root(N,E) \to \eta_c \in E\} \quad N_{\eta_c} = \{\eta_x \in N : \eta_c \leq_{E^*} \eta_x\} \\
E_{\eta_c} = \{\eta_x \to \eta_y \in E : \eta_x, \eta_y \in N_{\eta_c}\} \qquad \gamma_{\eta_c} = \{\eta_x \in \gamma : \eta_x \in N_{\eta_c}\}.
\end{gathered} \tag{34}$$

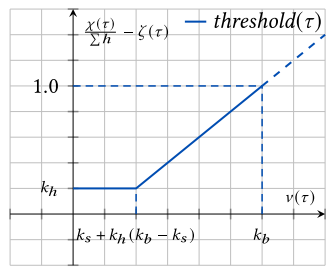
The Accept predicate. Accept predicate prevents the expansion of trails after they reach the size limit. We calculate the size of a trail according to the following equation:

$$v(N, E, \gamma) \equiv \sum_{\eta_{s_1,\ldots,s_n} \in N} codeSize(s_n). \tag{35}$$

Trail recursion is restricted with the function ζ that computes the sum of $2^{recursionDepth}$ of all the nodes, multiplied by a small, experimentally determined constant k_{rec} (in the following, $\delta_{x,y}$ is the Kronecker delta function, i.e., 1 when $x = y$ and 0 otherwise):

$$\zeta(N, E, \gamma) \equiv k_{rec} \cdot \sum_{\eta_{s_1,\ldots,s_n} \in N} 2^{recursionDepth(s_1,\ldots,s_n)} - 1 \quad where \quad recursionDepth(s_1, \ldots, s_n) = \sum_{i \in \{1,\ldots,n-1\}} \delta_{s_i, s_n} \tag{36}$$

As per Equation (37) the extended trail is accepted if its relative hotness decremented by the recursion penalty $\zeta(\tau)$ is larger than the *threshold* function. The threshold function depends on the size of the trail τ and is illustrated in the plot below. The threshold is a small, experimentally determined constant k_h if the trail size $v(\tau)$ is small—hence, small trails are almost always expanded. However, after the size $v(\tau)$ exceeds the value $k_s + k_h \cdot (k_b - k_s)$, the threshold starts rising linearly. A large trail can only be expanded if it is "very hot," but the likelihood of expansion eventually approaches zero as the size of that trail approaches the constant k_b.



$$\text{Accept}(\tau) \equiv \frac{\chi(\tau)}{\sum\limits_{(L,h) \in \Pi} h} - \zeta(\tau) > threshold(\tau) \quad where \quad threshold(\tau) = \max\left(k_h, \frac{v(\tau) - k_s}{k_b - k_s}\right) \tag{37}$$

Lemma D.4. *The left-hand side expression* $\frac{\chi(\tau)}{\sum\limits_{(L,h)\in\Pi} h} - \zeta(\tau)$ *from Equation* (37) *is asymptotically smaller than the the right-hand side expression* $threshold(\tau)$ *for any call graph* G *and profile set* Π.

Proof. For any call graph G, each call tree is either finite or infinite. If call trees are finite, then the trail size is bound (by Equation (7)), which imposes an upper bound on the $\chi(\tau)$ in the left-hand-side expression. If the call trees are infinite, then the trail size is not bound, but the call tree must contain recursive calls. The trail hotness $\chi(\tau)$ and size $v(\tau)$ grow linearly with each recursive call, but the penalty $\zeta(\tau)$ grows exponentially. Since the right-hand-side expression does not have any penalty, it is asymptotically larger than the left-hand side. □

Theorem D.5. *When executed with the hot-code detection policies from Equations* (30)–(32), *and* (37), *Algorithm 2 terminates for any input call graph* G *and set of profiles* Π.

Proof. We consider whether the repetitive steps in Algorithm 2 terminate. The operation fix $\overset{\textcircled{a}}{\nabla}_{root}$ terminates in a finite number of steps, as was already shown by Lemma 3.1.

This leaves us with the `while`-loop in line 4. To show that this loop eventually terminates, we show that the condition $T \setminus F = \varnothing$ from Equation (31) eventually holds. Consider the set $T \setminus F$. In each loop iteration, a trail τ is picked, and a set of *strictly larger* trails is created when extending τ across its calling contexts in line 7. The trail τ is then either excluded from T in line 10 or placed into F, meaning that it is no longer considered a top trail. We are left with a trail $T \setminus F$ that does not contain τ, but may instead contain a *subset of trails that are strictly larger than* τ. If the loop iterations are repeated sufficiently many times, then the Accept predicate from Equation (37) will

eventually be false for all trails in $T \setminus F$ as a consequence of Lemma D.4. Thus, eventually $T \setminus F$ stops growing. Moreover, this causes F to eventually start strictly growing, since each trail τ from T is eventually picked by TopTrail and moved to F. Therefore, the DetectionDone condition is eventually satisfied, and this terminates the algorithm. □

D.2 Hot-callee Classification Details

This section contains the definitions and details of the trail-match and trail-cut operations introduced in Section 3.6, which are omitted there for simplicity.

Definition of the *trail-matching operation* $\downarrow$ in Equation (38) says the following: Given a trail set T and a node $\eta_{s_1,\ldots,s_m,\ldots,s_n}$ of the inlining tree, find a trail that contains the longest suffix of the call sequence $s_1, \ldots, s_m, \ldots, s_n$.

$$T \downarrow \eta_{s_1,\ldots,s_m,\ldots,s_n} \equiv \underset{(N_\tau, E_\tau, \gamma_\tau) \in T, \exists \eta_{s_m,\ldots,s_n} \in N_\tau}{\arg\max} \; n - m \qquad \textit{such that} \quad \{(N_\tau, E_\tau, \gamma_\tau) \in T : \exists \eta_{s_m,\ldots,s_n} \in N_\tau\} \neq \varnothing \tag{38}$$

In the best case, the result of trail-matching is the same trail τ that the current hot compilation of s_1 starts with, i.e., $root(N_\tau, E_\tau) = \eta_{s_1}$. Otherwise, the matching between the inlining tree and the original trail τ is interrupted (indicating that some of the calls in the call sequence $s_1, \ldots, s_m, \ldots, s_n$ are cold), and some node of the inlining tree (which comes after that interruption) matches a trail that starts with the node η_{s_m} (indicating that $s_{m-1} \to s_m$ is a call from a cold to a hot subroutine).

A callee in the final inlining tree is considered hot if and only if there exists a matching trail:

$$\textsc{IsHot}(\eta) \equiv \exists \tau \in T, \tau = T \downarrow \eta. \tag{39}$$

To ensure that InlineHot from Section 3.5 works, we additionally need to associate a trail to each hot compilation. For the initial set of hot subroutines the association is straightforward, and for the transitive hot compilations, we use the *trail-cut operation*.

Trail-cut operation $\oslash$ is formally defined in Equation (40) Given a breadcrumb trail $\tau = (N_\tau, E_\tau, \gamma_\tau)$ and its node $\eta_{s_1,\ldots,s_m} \in N_\tau$, the operation cuts the trail in half, such that the resulting trail is the subtree starting at node $\eta_{s_1,\ldots,s_m}$.

$$\begin{aligned} (N_\tau, E_\tau, \gamma_\tau) \oslash \eta_{s_1,\ldots,s_m} &\equiv (N, E, \gamma) \quad \textit{where} \quad \eta_{s_1,\ldots,s_m} \in N_\tau \\ N &= \{\eta_{s_m,\ldots,s_n} : \eta_{s_1,\ldots,s_m,\ldots,s_n} \in N_\tau\} \quad \gamma = \{\eta_{s_m,\ldots,s_n} : \eta_{s_1,\ldots,s_m,\ldots,s_n} \in \gamma_\tau\} \\ E &= \{\eta_{s_m,\ldots,s_n} \to \eta_{s_m,\ldots,s_n,s_{n+1}} : \eta_{s_1,\ldots,s_m,\ldots,s_n} \to \eta_{s_1,\ldots,s_m,\ldots,s_n,s_{n+1}} \in E_\tau\} \end{aligned} \tag{40}$$

Callee trail. The trail for the hot callee $\eta_{s_1,\ldots,s_n}$ is determined with a combination of trail matching and trail cutting. First, we find a matching trail for $\eta_{s_1,\ldots,s_n}$ in the trail-set T, and we then cut that trail at the subtree that corresponds to the call sequence $s_1, \ldots, s_n$:

$$\textit{calleeTrail}(T, \eta_{s_1,\ldots,s_m,\ldots,s_n}) \equiv (T \downarrow \eta_{s_1,\ldots,s_m,\ldots,s_n}) \oslash \eta_{s_m,\ldots,s_n} \quad \textit{where} \quad \eta_{s_m,\ldots,s_n} \in (T \downarrow \eta_{s_1,\ldots,s_m,\ldots,s_n}) \,. \tag{41}$$

A hot compilation unit whose root subroutine corresponds to some callee η of the previous hot compilation unit is always associated with a trail determined by the expression $\textit{calleeTrail}(T, \eta)$.

REFERENCES

[1] Oracle Company. 2015. Java Virtual Machine Specification (Java SE 8 Edition): Chapter 4, the Class File Format.
[2] Free Software Foundation. 2018. GCC.
[3] Free Software Foundation. 2018. GCC 8 Changes.
[4] Free Software Foundation. 2018. GCC GENERIC.
[5] Free Software Foundation. 2018. GCC GIMPLE.
[6] Free Software Foundation. 2018. GCC RTL.
[7] LLVM Project. 2018. LLVM.

[8] LLVM Project. 2018. LLVM Cost-Benefit Estimation Implementation at GitHub. Retrieved from https://github.com/llvm-mirror/llvm/blob/88ab6705571782fa664ecfa71b2f959a0daf2d78/lib/Analysis/InlineCost.cpp
[9] LLVM Project. 2018. LLVM Inliner Implementation at GitHub. Retrieved from https://github.com/llvm-mirror/llvm/blob/6f1d64eb934e12ca5e8dcd378f88d1e6b80e8c55/lib/Transforms/IPO/Inliner.cpp
[10] LLVM Project. 2018. LLVM Inlining Parameters.
[11] LLVM Project. 2018. LLVM Language Reference Manual.
[12] Pavel Kosov and Sergey Yakushkin. 2020. LLVM PGO Instrumentation: Example of CallSite-Aware Profiling.
[13] Oracle Company. 2021. Control-flow-graph analysis in the Graal codebase. Retrieved December 7, 2021 from https://github.com/oracle/graal/blob/5708f348ad6a49511f0e3caf5314d72ca8c017e7/compiler/src/org.graalvm.compiler.nodes/src/org/graalvm/compiler/nodes/cfg/ControlFlowGraph.java
[14] Eclipse Foundation. 2021. Eclipse Test and Performance Tool Platform. Retrieved December 8, 2021 from http://archive.eclipse.org/archived%5C_projects/tptp.tgz
[15] Oracle Company. 2021. HotSpot Runtime Overview.
[16] LLVM Project. 2021. LLVM profile-guided optimizations. Retrieved December 6, 2021 from https://clang.llvm.org/docs/UsersManual.html#profile-guided-optimization
[17] Oracle Company. 2021. Netbeans: Open source Java profiler. v6.7.
[18] Oracle Company. 2021. OpenJDK 8 Optional Class.
[19] Oracle Company. 2021. Parallel Garbage Collector.
[20] IBM Corporation. 2021. Profile-Guided Optimization (PGO) using GCC.
[21] Oracle Company. 2021. Serial Native Image Garbage Collector.
[22] Oracle Company. 2021. Speculative guard motion in GraalVM. Retrieved May 27, 2021 from https://github.com/oracle/graal/commit/6dcc8e4a57d23e7aaf85eeb8dae7ef501b59c18b#diff1e4c4d8dd65775bb5c116be6e862315ddebf9b0d84aec949a79af159ef899df4
[23] Google LLC. 2021. V8 Engine.
[24] Martin Jambor, Jan Hubicka, Richard Biener, Martin Liska, Michael Matz, and Brent Hollingsworth. 2022. PGO in GCC 11. Retrieved from https://documentation.suse.com/sbp/server-linux/single-html/SBP-GCC-11/index.html#sec-gcc11-pgo
[25] George B. Dantzig. 1990. Origins of the simplex method. In *A History of Scientific Computing. Association for Computing Machinery*, New York, NY, 141–151. Retrieved from https://doi.org/10.1145/87252.88081
[26] Oracle Company. 2023. Java Mission Control.
[27] LLVM Project. 2023. LLVM PGO Context Sensitivity.
[28] Valgrind Developers. 2023. Valgrind.
[29] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. 2000. The Jalapeño virtual machine. *IBM Syst. J.* 39, 1 (Jan. 2000), 211–238. DOI: https://doi.org/10.1147/sj.391.0211
[30] Glenn Ammons, Thomas Ball, and James R. Larus. 1997. Exploiting hardware performance counters with flow and context sensitive profiling. *SIGPLAN Not.* 32, 5 (May 1997), 85–96. DOI: https://doi.org/10.1145/258916.258924
[31] Glenn Ammons, Jong-Deok Choi, Manish Gupta, and Nikhil Swamy. 2004. Finding and removing performance bottlenecks in large systems. In *Proceedings of the European Conference on Object-Oriented Programming*. Springer, 172–196.
[32] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. 2000. Adaptive optimization in the Jalapeño JVM. *SIGPLAN Not.* 35, 10 (Oct 2000), 47–65. DOI: https://doi.org/10.1145/354222.353175
[33] 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)*. ACM, New York, NY, 52–64. DOI: https://doi.org/10.1145/351397.351416
[34] Matthew Arnold, Michael Hind, and Barbara G. Ryder. 2002. Online feedback-directed optimization of Java. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*. Association for Computing Machinery, New York, NY, 111–129. DOI: https://doi.org/10.1145/582419.582432
[35] Matthew Arnold and Barbara G. Ryder. 2001. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*. Association for Computing Machinery, New York, NY, 168–179. DOI: https://doi.org/10.1145/378795.378832
[36] Matthew Arnold and Barbara G. Ryder. 2001. A framework for reducing the cost of instrumented code. *SIGPLAN Not.* 36, 5 (May 2001), 168–179. DOI: https://doi.org/10.1145/381694.378832
[37] Giorgio Ausiello, Camil Demetrescu, Irene Finocchi, and Donatella Firmani. 2012. K-calling context profiling. *SIGPLAN Not.* 47, 10 (Oct. 2012), 867–878. DOI: https://doi.org/10.1145/2398857.2384679

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

[39] Andrew Ayers, Richard Schooler, and Robert Gottlieb. 1997. Aggressive inlining. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'97)*. ACM, New York, NY, 134–145. DOI: https://doi.org/10.1145/258915.258928

[40] Ivan Baev. 2015. Profile-based Indirect Call Promotion. Retrieved from https://llvm.org/devmtg/2015-10/#talk3

[41] J. Eugene Ball. 1979. Predicting the effects of optimization on a procedure body. In *Proceedings of the SIGPLAN Symposium on Compiler Construction (SIGPLAN'79)*. ACM, New York, NY, 214–220. DOI: https://doi.org/10.1145/800229.806972

[42] Thomas Ball and James R. Larus. 1994. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.* 16, 4 (July 1994), 1319–1360. DOI: https://doi.org/10.1145/183432.183527

[43] Rajkishore Barik and Vivek Sarkar. 2009. Interprocedural load elimination for dynamic optimization of parallel programs. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT'09)*. IEEE Computer Society, 41–52. DOI: https://doi.org/10.1109/PACT.2009.32

[44] 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 (Oct. 2017), 27 pages. DOI: https://doi.org/10.1145/3133876

[45] Yosi Ben Asher, Omer Boehm, Daniel Citron, Gadi Haber, Moshe Klausner, Roy Levin, and Yousef Shajrawi. 2008. *Aggressive Function Inlining: Preventing Loop Blockings in the Instruction Cache*. Springer Berlin, 384–397. DOI: https://doi.org/10.1007/978-3-540-77560-7_26

[46] 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.

[47] Michael D. Bond and Kathryn S. McKinley. 2007. Probabilistic calling context. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'07)*. Association for Computing Machinery, New York, NY, 97–112. DOI: https://doi.org/10.1145/1297027.1297035

[48] Michael D. Bond and Kathryn S. McKinley. 2007. Probabilistic calling context. *SIGPLAN Not.* 42, 10 (Oct. 2007), 97–112. DOI: https://doi.org/10.1145/1297105.1297035

[49] Dries Buytaert, Andy Georges, Michael Hind, Matthew Arnold, Lieven Eeckhout, and Koen De Bosschere. 2007. Using HPM-sampling to drive dynamic compilation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'07)*. Association for Computing Machinery, New York, NY, 553–568. DOI: https://doi.org/10.1145/1297027.1297068

[50] Kevin Casey, David Gregg, M. Anton Ertl, and Andrew Nisbet. 2003. Towards superinstructions for Java interpreters. In *Software and Compilers for Embedded Systems*, Andreas Krall (Ed.). Springer Berlin, 329–343.

[51] Dhruva R. Chakrabarti and Shin-Ming Liu. 2006. Inline analysis: Beyond selection heuristics. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'06)*. IEEE Computer Society, Washington, DC, 221–232. DOI: https://doi.org/10.1109/CGO.2006.17

[52] Craig Chambers and David Ungar. 1991. Making pure object-oriented languages practical. In *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'91)*. Association for Computing Machinery, New York, NY, 1–15. DOI: https://doi.org/10.1145/117954.117955

[53] P. P. Chang and W.-W. Hwu. 1989. Inline function expansion for compiling C programs. *SIGPLAN Not.* 24, 7 (June 1989), 246–257. DOI: https://doi.org/10.1145/74818.74840

[54] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen-mei W. Hwu. 1992. Profile-guided automatic inline expansion for C programs. *Softw. Pract. Exper.* 22, 5 (May 1992), 349–369. DOI: https://doi.org/10.1002/spe.4380220502

[55] Pohua P. Chang, Scott A. Mahlke, and Wen-Mei W. Hwu. 1991. Using profile information to assist classic code optimizations. *Softw. Pract. Exper.* 21, 12 (Dec. 1991), 1301–1321. DOI: https://doi.org/10.1002/spe.4380211204

[56] Dehao Chen, Neil Vachharajani, Robert Hundt, Shih-Wei Liao, Vinodha Ramasamy, Paul Yuan, Wenguang Chen, and Weimin Zheng. 2010. Taming hardware event samples for FDO compilation. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'10)*. Association for Computing Machinery, New York, NY, 42–52. DOI: https://doi.org/10.1145/1772954.1772963

[57] Cliff Click. 1995. Global code motion/global value numbering. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*. ACM, New York, NY, 246–257. DOI: https://doi.org/10.1145/207110.207154

[58] Cliff Click and Michael Paleczny. 1995. A simple graph-based intermediate representation. In *Proceedings of the ACM SIGPLAN Workshop on Intermediate Representations (IR'95)*. ACM, New York, NY, 35–49. DOI: https://doi.org/10.1145/202529.202534

[59] Keith D. Cooper, Mary W. Hall, and Linda Torczon. 1992. Unexpected side effects of inline substitution: A case study. *ACM Lett. Program. Lang. Syst.* 1, 1 (Mar. 1992), 22–32. DOI: https://doi.org/10.1145/130616.130619

[60] Keith D. Cooper, Timothy J. Harvey, and Todd Waterman. 2008. An adaptive strategy for inline substitution. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction (CC'08/ETAPS'08)*. Springer-Verlag, Berlin, 69–84. Retrieved from http://dl.acm.org/citation.cfm?id=1788374.1788381

[61] George B. Dantzig, Alex Orden, and Philip Wolfe. 1955. The generalized simplex method for minimizing a linear form under linear inequality restraints. *Pacific Journal of Mathematics* 5, 2 (1955), 183–195.

[62] Jeffrey Dean and Craig Chambers. 1994. Towards better inlining decisions using inlining trials. In *Proceedings of the ACM Conference on LISP and Functional Programming (LFP'94)*. ACM, New York, NY, 273–282. DOI: https://doi.org/10.1145/182409.182489

[63] David Detlefs and Ole Agesen. 1999. *Inlining of Virtual Methods*. Springer Berlin, 258–277. DOI: https://doi.org/10.1007/3-540-48743-3_12

[64] M. Dmitriev. 2004. Selective profiling of Java applications using dynamic bytecode instrumentation. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'04)*. 141–150. DOI: https://doi.org/10.1109/ISPASS.2004.1291366

[65] Gilles Duboscq, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Speculation without regret: Reducing deoptimization meta-data in the graal compiler. In *Proceedings of the International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ'14)*. Association for Computing Machinery, New York, NY, 187–193. DOI: https://doi.org/10.1145/2647508.2647521

[66] 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, New York, NY, 1–10. DOI: https://doi.org/10.1145/2542142.2542143

[67] 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)*. ACM, New York, NY, 1–10. DOI: https://doi.org/10.1145/2542142.2542143

[68] Evelyn Duesterwald and Vasanth Bala. 2000. Software profiling for hot path prediction: Less is more. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'00)*. Association for Computing Machinery, New York, NY, 202–211. DOI: https://doi.org/10.1145/378993.379241

[69] Josef Eisl, Matthias Grimmer, Doug Simon, Thomas Würthinger, and Hanspeter Mössenböck. 2016. Trace-based register allocation in a JIT compiler. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ'16)*. Association for Computing Machinery, New York, NY, Article 14, 11 pages. DOI: https://doi.org/10.1145/2972206.2972211

[70] Josef Eisl, Stefan Marr, Thomas Würthinger, and Hanspeter Mössenböck. 2017. Trace register allocation policies: Compile-time vs. performance trade-offs. In *Proceedings of the 14th International Conference on Managed Languages and Runtimes (ManLang'17)*. ACM, New York, NY, 92–104. DOI: https://doi.org/10.1145/3132190.3132209

[71] Stephen J. Fink and Feng Qian. 2003. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (CGO'03)*. IEEE Computer Society, 241–252.

[72] Olivier Flückiger, Andreas Wälchli, Sebastián Krynski, and Jan Vitek. 2020. Sampling optimized code for type feedback. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages (DLS'20)*. Association for Computing Machinery, New York, NY, 99–111. DOI: https://doi.org/10.1145/3426422.3426984

[73] Edward Fredkin. 1960. Trie memory. *Commun. ACM* 3, 9 (Sep. 1960), 490–499. DOI: https://doi.org/10.1145/367390.367400

[74] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically rigorous Java performance evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'07)*. Association for Computing Machinery, New York, NY, 57–76. DOI: https://doi.org/10.1145/1297027.1297033

[75] James Gosling. 1995. Java intermediate bytecodes: ACM SIGPLAN Workshop on Intermediate Representations (IR'95). *SIGPLAN Not.* 30, 3 (Mar. 1995), 111–118. DOI: https://doi.org/10.1145/202530.202541

[76] David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers. 1995. Profile-guided receiver class prediction. In *Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*. Association for Computing Machinery, New York, NY, 108–123. DOI: https://doi.org/10.1145/217838.217848

[77] Martin Hirzel and Trishul Chilimbi. 2001. Bursty tracing: A framework for low-overhead temporal profiling. In *Proceedings of the 4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO'01)*. Citeseer, 117–126.

[78] Urs Hölzle and David Ungar. 1994. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94)*. ACM, New York, NY, 326–336. DOI: https://doi.org/10.1145/178243.178478

[79] Jan Hubicka. 2005. Profile driven optimisations in GCC. In *GCC Summit Proceedings*. Citeseer, 107–124.

[80] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the future: The story of squeak, a practical smalltalk written in itself. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'97)*. Association for Computing Machinery, New York, NY, 318–326. DOI: https://doi.org/10.1145/263698.263754

[81] Suresh Jagannathan and Andrew Wright. 1996. Flow-directed inlining. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'96)*. ACM, New York, NY, 193–205. DOI: https://doi.org/10.1145/231379.231417

[82] Thomas Kistler and Michael Franz. 2003. Continuous program optimization: A case study. *ACM Trans. Program. Lang. Syst.* 25, 4 (July 2003), 500–548. DOI: https://doi.org/10.1145/778559.778562

[83] Donald E. Knuth. 1971. An empirical study of FORTRAN programs. *Softw. Pract. Exp.* 1, 2 (1971), 105–133. DOI: https://doi.org/10.1002/spe.4380010203

[84] Thomas Kotzmann and Hanspeter Mossenbock. 2007. Run-time support for optimizations based on escape analysis. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'07)*. IEEE Computer Society, Washington, DC, 49–60. DOI: https://doi.org/10.1109/CGO.2007.34

[85] C. Krintz. 2003. Coupling on-line and off-line profile information to improve program performance. In *International Symposium on Code Generation and Optimization (CGO'03)*. 69–78. DOI: https://doi.org/10.1109/CGO.2003.1191534

[86] Aditya Kumar. 2019. Hot Cold Splitting Optimization Pass In LLVM. Retrieved from https://llvm.org/devmtg/2019-10/talk-abstracts.html#tech8

[87] Anatole Le, Ondřej Lhoták, and Laurie Hendren. 2005. Using inter-procedural side-effect information in JIT optimizations. In *Compiler Construction*, Rastislav Bodik (Ed.). Springer Berlin, 287–304.

[88] David Leopoldseder, Lukas Stadler, Thomas Würthinger, Josef Eisl, Doug Simon, and Hanspeter Mössenböck. 2018. Dominance-based duplication simulation (DBDS): Code duplication to enable compiler optimizations. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'18)*. ACM, New York, NY, 126–137. DOI: https://doi.org/10.1145/3168811

[89] Roy Levin, Ilan Newman, and Gadi Haber. 2008. Complementing missing and inaccurate profiling using a minimum cost circulation algorithm. In *Proceedings of the 3rd International Conference on High Performance Embedded Architectures and Compilers (HiPEAC'08)*. Springer-Verlag, Berlin, 291–304.

[90] Matthew Arnold and Peter Sweeney. 2000. *Approximating the Calling Context Tree Via Sampling*. IBM.

[91] Scott Milton and Heinrich (Heinz) Schmidt. 1994. Dynamic Dispatch in Object-Oriented Languages. (March 1994).

[92] Anders Møller and Oskar Haarklou Veileborg. 2020. Eliminating abstraction overhead of Java stream pipelines using ahead-of-time program optimization. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 168 (Nov. 2020), 29 pages. DOI: https://doi.org/10.1145/3428236

[93] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2010. Evaluating the accuracy of Java profilers. *ACM SIGPLAN Not.* 45, 6 (2010), 187–197.

[94] Diego Novillo. 2014. SamplePGO: The power of profile guided optimizations without the usability burden. In *Proceedings of the LLVM Compiler Infrastructure in HPC (LLVM-HPC'14)*. IEEE Press, 22–28.

[95] Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java HotSpot server compiler. In *Proceedings of the Symposium on JavaTM Virtual Machine Research and Technology Symposium (JVM'01)*. USENIX Association, Berkeley, CA, 1–1. Retrieved from http://dl.acm.org/citation.cfm?id=1267847.1267848

[96] Simon Peyton Jones and Simon Marlow. 2002. Secrets of the Glasgow Haskell compiler inliner. *J. Funct. Program.* 12, 5 (July 2002), 393–434. DOI: https://doi.org/10.1017/S0956796802004331

[97] Aleksandar Prokopec, Gilles Duboscq, David Leopoldseder, and Thomas Würthinger. 2019. An optimization-driven incremental inline substitution algorithm for just-in-time compilers. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO'19)*. IEEE Press, Piscataway, NJ, 164–179. Retrieved from http://dl.acm.org/citation.cfm?id=3314872.3314893

[98] Aleksandar Prokopec, David Leopoldseder, Gilles Duboscq, and Thomas Würthinger. 2017. Making collection operations optimal with aggressive JIT compilation. In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala (SCALA'17)*. ACM, New York, NY, 29–40. DOI: https://doi.org/10.1145/3136000.3136002

[99] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tuma, 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 ACM SIGPLAN Conference on Programming Language*

Design and Implementation, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 31–47. Retrieved from http://dblp.uni-trier.de/db/conf/pldi/pldi2019.html#ProkopecRLD0SBZ19

[100] Rodric M. Rabbah, Hariharan Sandanagobalane, Mongkol Ekpanyapong, and Weng-Fai Wong. 2004. Compiler orchestrated prefetching via speculation and predication. *SIGPLAN Not.* 39, 11 (Oct. 2004), 189–198. DOI: https://doi.org/10.1145/1037187.1024416

[101] Steven P. Reiss and Manos Renieris. 2001. Encoding program executions. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE'01)*. IEEE, 221–230.

[102] Alan D. Samples. 1991. *Profile-driven Compilation*. Technical Report. University of California, Berkeley, Department of Electrical Engineering and Computer Sciences.

[103] Aibek Sarimbekov, Philippe Moret, Walter Binder, Andreas Sewe, and Mira Mezini. 2011. Complete and platform-independent calling context profiling for the Java virtual machine. *Electron. Notes Theor. Comput. Sci.* 279, 1 (2011), 61–74. DOI: https://doi.org/10.1016/j.entcs.2011.11.006

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

[105] Manuel Serrano. 1997. *Inline Expansion: When and How?* Springer Berlin, 143–157. DOI: https://doi.org/10.1007/BFb0033842

[106] Mauricio Serrano and Xiaotong Zhuang. 2009. Building approximate calling context from partial call traces. In *Proceedings of the International Symposium on Code Generation and Optimization*. 221–230. DOI: https://doi.org/10.1109/CGO.2009.12

[107] Andreas Sewe, Jannik Jochem, and Mira Mezini. 2011. Next in line, please!: Exploiting the indirect benefits of inlining by accurately predicting further inlining. In *Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11 (SPLASH'11 Workshops)*. ACM, New York, NY, 317–328. DOI: https://doi.org/10.1145/2095050.2095102

[108] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. 2011. Da capo con scala: Design and analysis of a scala benchmark suite for the Java virtual machine. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'11)*. 657–676.

[109] Denys Shabalin. 2020. *Just-In-Time Performance Without Warm-Up*. IINFCOM, Lausanne, 165. http://infoscience.epfl.ch/record/275338

[110] Denys Shabalin and Martin Odersky. 2018. Interflow: Interprocedural flow-sensitive type inference and method duplication. In *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala (Scala'18)*. Association for Computing Machinery, New York, NY, 61–71. DOI: https://doi.org/10.1145/3241653.3241660

[111] Doug Simon, Christian Wimmer, Bernhard Urban, Gilles Duboscq, Lukas Stadler, and Thomas Würthinger. 2015. Snippets: Taking the high road to a low level. *ACM Trans. Archit. Code Optim.* 12, 2, Article 20 (June 2015), 25 pages. DOI: https://doi.org/10.1145/2764907

[112] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Partial escape analysis and scalar replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'14)*. ACM, New York, NY, Article 165, 10 pages. DOI: https://doi.org/10.1145/2544137.2544157

[113] Codruţ Stancu, Christian Wimmer, Stefan Brunthaler, Per Larsen, and Michael Franz. 2014. Comparing points-to static analysis with runtime recorded profiling data. In *Proceedings of the International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ'14)*. Association for Computing Machinery, New York, NY, 157–168. DOI: https://doi.org/10.1145/2647508.2647524

[114] Edwin Steiner, Andreas Krall, and Christian Thalinger. 2007. Adaptive inlining and on-stack replacement in the CACAO virtual machine. In *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java (PPPJ'07)*. ACM, New York, NY, 221–226. DOI: https://doi.org/10.1145/1294325.1294356

[115] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. 2002. An empirical study of method inlining for a Java just-in-time compiler. In *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium*. USENIX Association, 91–104.

[116] Omri Traub, Stuart Schechter, and Michael D. Smith. 2000. *Ephemeral Instrumentation for Lightweight Program Profiling*. Unpublished Technical Report, Department of Electrical Engineering and Computer Science, Harvard University, Cambridge, MA.

[117] April W. Wade, Prasad A. Kulkarni, and Michael R. Jantz. 2017. AOT vs. JIT: Impact of profile data on code quality. *SIGPLAN Not.* 52, 5 (June 2017), 1–10. DOI: https://doi.org/10.1145/3140582.3081037

[118] Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.* 13, 2 (Apr. 1991), 181–210. DOI: https://doi.org/10.1145/103135.103136

[119] John Whaley. 2000. A portable sampling-based profiler for Java virtual machines. In *Proceedings of the ACM Conference on Java Grande (JAVA'00)*. Association for Computing Machinery, New York, NY, 78–87. DOI: https://doi.org/10.1145/337449.337483

[120] 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: https://doi.org/10.1145/3360610

[121] Thomas Wuerthinger, Christian Wimmer, and Hanspeter Moessenboeck. 2008. Visualization of program dependence graphs. *Compiler Construction*, L. Hendren (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 193–196.

[122] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical partial evaluation for high-performance dynamic language runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'17)*. ACM, New York, NY, 662–676. DOI: https://doi.org/10.1145/3062341.3062381

[123] Graham Yiu. 2017. Partial Inlining with Multi-region Outlining based on PGO Information. https://reviews.llvm.org/D38190

[124] Peng Zhao and José Nelson Amaral. 2004. *To Inline or Not to Inline? Enhanced Inlining Decisions*. Springer Berlin, 405–419. DOI: https://doi.org/10.1007/978-3-540-24644-2_26

Received 21 January 2022; revised 1 July 2023; accepted 17 July 2023