An Optimization-Driven Incremental Inline Substitution Algorithm for Just-in-Time Compilers

Aleksandar Prokopec
Oracle Labs, Switzerland
aleksandar.prokopec@gmail.com

Gilles Duboscq
Oracle Labs, Switzerland
gilles.m.duboscq@oracle.com

David Leopoldseder
JKU Linz, Austria
david.leopoldseder@jku.at

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

Abstract—Inlining is one of the most important compiler optimizations. It reduces call overheads and widens the scope of other optimizations. But, inlining is somewhat of a black art of an optimizing compiler, and was characterized as a computationally intractable problem. Intricate heuristics, tuned during countless hours of compiler engineering, are often at the core of an inliner implementation. And despite decades of research, well-established inlining heuristics are still missing.

In this paper, we describe a novel inlining algorithm for JIT compilers that incrementally explores a program’s call graph, and alternates between inlining and optimizations. We devise three novel heuristics that guide our inliner: adaptive decision thresholds, callsite clustering, and deep inlining trials. We implement the algorithm inside Graal, a dynamic JIT compiler for the HotSpot JVM. We evaluate our algorithm on a set of industry-standard benchmarks, including Java DaCapo, Scalabench, Spark-Perf, STM Bench7 and other benchmarks, and we conclude that it significantly improves performance, surpassing state-of-the-art inlining approaches with speedups ranging from 5% up to 3x.

Index Terms—just-in-time compilation, inlining, polymorphic dispatch, cost-benefit analysis, optimization-driven inlining, priority-based inlining, inline substitution, inline expansion

I. INTRODUCTION

The inline substitution replaces callsites with the bodies of the respective callees [74]. In a compiler that relies mainly on intraprocedural analysis, the benefit of inlining is the enabling of other optimizations [18, 36, 47]. Although, this transformation is straightforward, deciding which methods to inline requires intricate heuristics – Jones and Marlow called inlining a "black art" [47]. While a simplified formulation of inlining can be reduced to the Knapsack problem [74], its main difficulty is estimating the time savings from inlining a callee. Most inlining algorithms follow a similar pattern: use a heuristic to assign a benefit to each callsite, and inline them priority-wise until a budget limit is hit [13, 6, 8]. Optimization prediction is one way of assessing the benefits, which predicts the execution time savings by simulating the optimizations triggered by inlining – this was done in Algol [7] and Scheme [18], for example. The second way used to estimate benefits is by profiling callsite frequencies, as in CLU, C and Java, to name a few examples [74, 13, 33, 32]. We give a more detailed overview of related work in Section VI.

In this paper, we show how to improve existing inlining techniques for JIT compilers by proposing a novel online (as defined in Section II) inlining algorithm, which improves optimization prediction by incrementally exploring and specializing the program’s call tree, and clusters the callsites before inlining. We report the following new observations:

- In many programs, there is an impedance between the logical units of code (i.e. subroutines) and the optimizeable units of code (groups of subroutines). It is therefore beneficial to inline specific clusters of callsites, instead of a single callsite at a time. We describe how to identify such callsite clusters.
- When deciding when to stop inlining, using an adaptive threshold function (rather than a fixed threshold value) produces more efficient programs.
- By propagating a callsite’s argument types throughout the call tree, and then performing optimizations in the entire call tree, estimation of the execution time savings due to inlining becomes more accurate. These optimizations can improve the type precision at the other callsites, so this gets repeated until reaching a fixpoint. A callsite records its optimizations, hereby calculating its benefit. We call this deep inlining trials.

Based on these observations we derived a novel inlining algorithm and implemented it in the Graal compiler [20, 89, 64]. The contributions in this paper are as follows:

- We present an online inlining algorithm, which inlines clusters of related callsites in a priority order, and makes inlining decisions based on a callsite’s benefit and cost (Section III and Section IV). The benefit is estimated from a callsite’s execution frequency and its optimization potential, determined with a new technique named deep inlining trials. The inlining decisions rely on adaptive threshold functions (Section IV).
- We implemented the algorithm inside a production quality dynamic JIT compiler called Graal, which is a replacement for HotSpot’s C2 optimizing compiler. We present heuristics and techniques that we used to tune the quality of the inlining decisions (Section IV).
- We evaluated the algorithm with the Java DaCapo [9], Scala DaCapo [78], and several other benchmark suites in terms of the performance of the generated code. We compare the results against the HotSpot’s C2 compiler as well as the inlining algorithm used in the open-source version of Graal [33], and we report performance

\[1\] Artifact available at: https://doi.org/10.5281/zenodo.2328430
improvements ranging from 5% up to 3×. We present a performance breakdown of the heuristics used in the algorithm, showing that each component in our algorithm (cluster detection, deep optimization prediction, and adaptive threshold functions) improves the performance of the generated code (Section V).

II. PROBLEM STATEMENT

Scheifler showed that the inlining problem is NP-hard by reducing it to the Knapsack problem [74], where the Knapsack items represent the methods in a program, the weight of each item is the method size, and the value is the decrease in the runtime (i.e., benefit). The online inlining problem that we formulate here is more complicated because it only has access to a subset of call sites at any given point in time.

Online inlining problem. Given a set $\Pi$ of program methods $M$, a stream of compilation requests for a subset $C$ of those methods, and a maximum program size $S$, the objective is to assign a set of inlining decisions to each request in a way that minimizes the overall execution time. The requests from the stream $C$ can arrive at any time during the execution (before being compiled, the methods are interpreted). Scheifler’s inlining problem [74] is a special case in which the set of requests $C$ is the set of all methods $\Pi$, and in which the compilations run before the program starts.

Notably, an online inlining algorithm makes decisions in each individual method separately. Also, unlike a link-time algorithm, an online inlining algorithm does not have access to the complete call graph, it does not know which methods will be compiled in the future, nor which are the best inlining opportunities in future compilation requests.

Practical difficulties. Online inlining has several additional challenges. Our previous definition assumes that the method size and execution time can be accurately predicted, and that the future compilation requests are independent of the earlier decisions. Another indirect assumption was that more inlining always results in more runtime savings. These assumptions are not true in practice, as we argue next.

1. Noisy estimates: An inlining algorithm reasons about the reduction in a program’s execution time and the increase of the generated code. The runtime profiling information, such as the branch execution probabilities or loop backedge counters [13, 5], is used to estimate savings, but these hints are neither precise nor an accurate predictor of future behavior, due to type profile pollution [73], or phase shifts [37].

2. Compilation impact: If a call site $C_j$ to a method $M$ is inlined, runtimes such as the JVM stop measuring the hotness of $M$ at the call site $C_j$. This can prevent the compilation of $M$. Second, inlining decisions correlate with the compilation latency, which impacts the arrival of other requests. A delayed request will have more profiling information, making other profile-guided optimizations behave differently.

3. Non-linearity: Excessive inlining can put more pressure on limited hardware resources, such as the instruction cache [44], and degrade performance. Importantly, later optimizations with a limited budget are less effective if inlining produces a huge

III. ALGORITHM DESCRIPTION

In this section, we give a conceptual, high-level overview of the algorithm. We summarize the heuristics that guide the algorithm in Section V. The proposed online inlining algorithm takes advantage of three crucial observations. First, there is a discrepancy between the subroutines, which are the logical units, and groups of subroutines that call each other, which are the optimizable units. Therefore, the new algorithm uses a heuristic (shown later, in Listing 5) to identify such groups of subroutines, and either inline them together, or not at all.

Second, most inliners use a fixed threshold to decide when to stop inlining. However, this is detrimental if there are a few hot calls to small methods when the method runs out of budget. Consider, for example, the Scala program shown in Figure 1 – inlining the foreach call into the log method is only beneficial if the get and length calls inside the corresponding loop are also inlined. Thus, the proposed algorithm uses adaptive thresholds to allow the inlining of hot methods even if it is low on budget, as shown later in Section IV.

Finally, inlining decisions can be improved by propagating constants and type information throughout the call tree, and by speculatively triggering the optimizations before inlining the callees. Therefore, the proposed inliner alternates between call tree exploration, call tree simplification and inlining. We start by giving a high-level overview of this process.

High-level overview. A compilation starts an independent instance of our algorithm, whose high-level pseudocode is shown in Listing 1. The request consists of the intermediate representation of a method $\mu$, called the root compilation method [40, 22, 88]. The intuition that more inlining produces faster programs is thus correct only up to a certain limit, as we show in Section V.

Fig. 1: An Example Scala Program

<table>
<thead>
<tr>
<th>Listing 1: Incremental Inlining Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>input</strong>: method $\mu$ with inlined call sites</td>
</tr>
<tr>
<td><strong>output</strong>: method $\mu$</td>
</tr>
<tr>
<td>1 root = createRoot($\mu$);</td>
</tr>
<tr>
<td>2 while detectTermination(root) do</td>
</tr>
<tr>
<td>3 expand(root);</td>
</tr>
<tr>
<td>4 analyze(root);</td>
</tr>
<tr>
<td>5 inline(root);</td>
</tr>
</tbody>
</table>

1 def main(args: Array[String]) { |
2 async { log(getStackTrace) } |
3 log(args) |
4 } |
5 def log[T](xs: Array[T]) { |
6 xs.foreach(println) |
7 } |
8 trait IndexedSeqOptimized[T] { |
9 def get(i: Int): T |
10 def length: Int |
11 def foreach(f: T => Unit) { |
12 var i = 0 |
13 while (i < this.length) |
14 { f.apply(this.get(i)); i += 1 } |
15 } |
16 }

165
method. The algorithm creates the root call tree node for \( \mu \), which contains its intermediate representation and the list of child nodes. Each child corresponds to a callsite in \( \mu \). At this point, a child does not yet contain its respective body.

The expand step then partially expands the call tree. After that, the analyze step decides which parts of the call tree should be inlined. Finally, the inline step inlines those parts into the root method \( \mu \). These three phases, expansion, analysis and inlining, repeat until the algorithm terminates. In the following sections, we explain these phases in detail.

Example. Consider the Scala method main in Figure 1 which starts an asynchronous execution thread with async, and then calls log to output the command-line arguments. The log method uses the foreach method defined on the type IndexedSeqOptimized from the Scala standard library to print the array elements. The foreach is very general – it traverses the elements of any sequence collection with a while-loop and calls the specified function on each of the elements. The foreach method calls the length method, the get method to obtain the elements, and the apply to invoke the function \( f \) – all calls are polymorphic.

The call tree in Figure 2 initially has the root node main. The tag \( E \) means that the node was expanded. Callsites async, $anon and log are the child nodes with the annotation \( C \), which indicates that they are not yet expanded. The $anon node is a constructor for the lambda object passed to the async call. The algorithm then expands the nodes $anon and log, producing the call tree on the right. Since the implementation of the async method is not known in this example, the inliner assigns the annotation \( G \) to that call node. Finally, the algorithm analyzes the call tree and decides to inline the $anon callsite, as shown on the bottom. Unlike some alternative inlining algorithms [6, 5, 8, 22], our algorithm does not yet decide to inline foreach – the benefit of inlining foreach only becomes apparent once the call tree is explored more.

A. Call Tree Data Structure

Each call tree node represents a callsite to a method \( M_i \) in its parent, and its children represent the callsites \( C_j \) within the body of the method \( M_i \), as illustrated in Figure 2. The call tree is partial, meaning that it consists of the method \( \mu \) and a connected subset of the nodes reachable from \( \mu \).

Call tree nodes. The node data type is shown in Listing 2. Each node has a kind tag, which can be \( E \), indicating an expanded node, or \( C \), indicating a non-expanded cutoff node. In addition, a node can be tagged with \( D \), indicating that there was a callsite, but it was deleted by an optimization. Finally, \( G \) indicates that a particular callsite cannot be inlined. Each node except the root holds a pointer to the respective callsite in the parent’s body, and an expanded node holds the intermediate representation \( ir \) of the respective method.

Rationale. Many inlining algorithms use a complete call graph to represent the call hierarchy \( [4, 5, 6, 12, 13, 17, 35, 85, 22] \), which consists of all the methods in the program, and directed edges representing the callsites in those methods. Our algorithm uses a partial call tree for three reasons:

1. Dynamic classloading: Dynamic runtimes such as the JVM can load code during the execution \( [41] \), so it is not possible to statically construct the complete call graph, as code might have never been executed and thus not eagerly loaded.
2. JIT compilation constraints: A dynamic compiler has a limited time budget, since it runs concurrently with the user code. Creating the complete call graph is expensive \( [21, 22] \).
3. Callsite specialization: A callsite within the root method \( \mu \) can be specialized with the callsite arguments in \( \mu \)'s body. Moreover, argument type information can be propagated through the call tree, allowing the specialization of each callsite. This is harder with a complete call graph, where each node represents the target of many callsites. Callsite specialization enables more accurate predictions about the costs and the benefits, as we show in Section V.

B. Expansion Phase

During the expansion, our algorithm heuristically chooses a cutoff node, and attempts to expand it. If a cutoff can be expanded, then the IR of the corresponding method is attached. For each callsite in the IR of the newly expanded node, a child cutoff node gets added. This is repeated until heuristically deciding that the tree is sufficiently expanded.

Description. The expand procedure in Listing 3 initializes a mapping queue between each node and the set of its children that should be considered for expansion. The expand then repeatedly descends into the call tree, until the expansionDone heuristic interrupts it. The descend procedure selects a path from the root to a cutoff node. To choose a subtree, descend heuristically picks the best child on the node’s queue, and ensures that a child \( c \) is kept on the queue of its parent \( n \) only if \( c \)'s queue is non-empty or \( c \) is a cutoff node.

Once descend finds a cutoff, it invokes the expandCutoff heuristic, which either expands the cutoff, or returns null to indicate that the cutoff should be left as-is.
Example. Consider the main method from Figure 1. At the start of the second expansion phase, the queues of the expanded nodes contain the only child, as shown below in the first call tree. The descend procedure picks the path to the foreach call, which is the only cutoff node.

In the second call tree, the queue of the newly expanded foreach node contains nodes 7, 6 and 5, corresponding to calls to apply, get and length, respectively. The policy gives a higher priority to the node 7 (for example, because that apply call appears in a loop), so it appears earlier. However, after the cutoff node 7 gets expanded, the policy decides that the priority of the node 7 should decrease, and now node 6 appears earlier, as shown in the third call tree.

C. Cost-Benefit Analysis Phase

The cost-benefit analysis determines the cost and the benefit of each node in the partial call tree, as well as the relationships between the methods that should be inlined together. For example, the analysis may decide that if a particular callee M is inlined, then its children must also be inlined.

Description. The cost-benefit analysis is a bottom-up traversal of the call tree. The policy-based analyzeNode procedure inspects the node’s intermediate representation and its children to estimate its benefit and its cost. This procedure may recursively replace the parameters in the method body with the callsite arguments, and trigger optimizations.

Example. Our implementation assigns cost-benefit tuples to nodes, but it also detects the clusters of nodes that should be inlined either together, or not at all. Consider the last call tree in the example from Section III-B. After the analysis, the nodes 3, 4, 5, 6 and 7 form one cluster (for example, because they simplify the loop in foreach when compiled together), as indicated with the dashed line in Figure 3. This means that they can either be inlined together or not at all. Separately, the node 8 comprises a second callsite cluster. The root node of each callsite is assigned a cost-benefit tuple that models the benefit (i.e. the estimated code size increase) of inlining the entire respective cluster. To avoid cluttering Figure 3 we omit concrete cost-benefit tuple assignments.

D. Inlining Phase

The goal of this phase is to inline clusters of related methods into the root compilation method. The inline method first creates a queue that initially contains the root’s children. As long as the queue is not empty, the bestCluster heuristic selects and removes a node from the queue that represents a cluster. If the canInline heuristic decides that the cluster can be inlined, then inlineCluster traverses the nodes of the cluster, and inlineIR replaces the callsite with the body of that cluster. The descendants of the cluster are put on the queue, and the loop repeats.
Example. Figure [5] shows two clusters, and the call trees after two calls to the inlineCluster method. Generally, the inlining can end earlier if the canInline heuristic decides that it ran out of budget. In this particular case, the inlining ends once the queue becomes empty.

IV. IMPLEMENTATION

Having shown the high-level structure of our inlining algorithm in Section III, we focus on the details of our implementation, which was done in the enterprise edition of the Graal JIT compiler. We explain several additional optimizations, and the implementations of heuristics that were shown with slanted typeface in Listings [1] [3] [4] and [5].

The motivations behind our heuristics are as follows. (1) We want to inline parts of the call tree with a high benefit, but spend as little budget as possible. (2) However, we avoid exploring one part of the call tree too much at the expense of other parts, even if it seems extremely beneficial – infrequently invoked methods can reveal type information that allows simplifying the hot methods. (3) When inlining a method M, we also want to inline other methods that M would inline if it were the compilation root. This prevents wasting the optimizations that would otherwise be triggered.

We first introduce the metrics tracked by our algorithm. First, we note that Graal can access the JVM profiling data [4] [38], such as branch probabilities, back-edge counters and receiver profiles. This allows computing, for each call node n, the call frequency f(n) relative to the root method.

Next, for each call subtree of a node n, we maintain the sum Srin(n) of all the call node IR sizes in that subtree.

\[ S_{rin}(n) = \sum_{m \in \text{subtree}(n)} |ir(m)| \]

For each call subtree below a call node n, we track the total size Sb(n) of all the IRs in all its cutoff nodes.

\[ S_b(n) = \sum_{m \in \text{subtree}(n), \text{kind}(m) = C} |ir(m)| \]

Similarly, we track the cutoff count Nc(n) in every subtree:

\[ N_c(n) = |\{m \in \text{subtree}(n) : \text{kind}(m) = C\}| \]

For a particular call site n with a frequency f(n) relative to the root, let Ns(n) be the number of arguments whose type is more concrete than the formal parameters, and Ns(n) be the number of simple optimizations triggered in the callee as a result of deep inlining trials. To estimate the local benefit B_L(n) of inlining the callee n, we use the following formula:

\[ B_L(n) = \begin{cases} f(n) \cdot (1 + N_s(n)) & \text{kind}(n) = C \\ f(n) \cdot (1 + N_s(n)) & \text{kind}(n) = E \end{cases} \]

Note that, in our implementation, we calculate Ns(n) only for the simplest optimizations, such as constant folding, strength reduction or branch pruning, and give them all equal weight. Refining this could yield more precise inlining decisions, but we did not investigate this in the current work.

Expansion. Recall the highestPriorityNode heuristic from the Listing[3] Its goal is to predict the nodes whose inlining reduces the execution time most, and which are at the same time likely to be inlined. We therefore assign priorities, and define the intrinsic priority P1(n) for a node n as:

\[ P_1(n) = \begin{cases} \frac{B_L(n)}{|ir(n)|} & \text{kind}(n) = C \\ \max_{c \in \text{children}(n)} P_1(c) & \text{kind}(n) = E \end{cases} \]

For a cutoff node, the intrinsic priority is a ratio between the benefit and the code size increase if it gets inlined. For an expanded node, priority P(n) is the maximum of the intrinsic priorities of its children. The final priority P(n) of a call node is the intrinsic priority P1(n) decreased by the penalty ψ(n):

\[ P(n) = P_1(n) - ψ(n) \]

We use ψ(n) to reduce the priority of the heavily explored subtrees. This prevents endlessly exploring a particular subtree that has some path in the call hierarchy with a high priority (e.g. a recursive method with a loop), while exploring another part of the call tree could be more beneficial (e.g. because it deletes the recursive call).

\[ ψ(n) = p_1 \cdot S_{rin}(n) + p_2 \cdot S_b(n) - b_1 \cdot \max(0, b_2 - N^2_c(n)) \]

The penalty ψ correlates with Srin and Sb, but is decreased if the subtree has only a few cutoffs left. The reasoning is that, even if the subtree is huge, it is likely that exploring few cutoffs would turn the entire subtree into a single cluster. We experimentally tuned Graal to use the values p1 = 10^-3, p2 = 10^-4, b1 = 0.5, and b2 = 10, as they generated the best results for our benchmarks in Section V. We believe that these parameters depend on the compiler implementation.

The expandCutoff heuristic decides whether to explore a cutoff node using the following formula:

\[ \frac{B_L(n)}{|ir(n)|} \geq e^{(S_{rin(root)} - r_1)/r_2} \]

The intuition is that the relative benefit threshold rises steadily as there are more and more nodes in the root method. The exponential function grows fast with call tree size, but it is smooth – if there are a few very beneficial calls after the typical tree size is exceeded, then it is still sensible to explore them. We experimentally determined that the values r1 ≈ 3000 and r2 ≈ 500 work well in our implementation.

Analysis. Our algorithm tracks the cost-benefit tuple b|c of each node. One of the tuple operations is merging ⊕:

\[ b_1|c_1 \oplus b_2|c_2 \equiv b_1 + b_2|c_1 + c_2 \]

The second tuple operation is comparison ⊙:

\[ b_1|c_1 \oslash b_2|c_2 \iff \frac{b_1}{c_1} \geq \frac{b_2}{c_2} \]

Finally, we define the cost to benefit ratio 〈·,·〉 as follows:

\[ (b|c| \equiv b/c \]

Our algorithm also tracks a mapping inlined from each node n to a boolean indicating if n is in the same cluster as its parent, and another mapping front that contains the descendants of the node that are not in the same cluster.

We implemented the analyzeNode heuristic as shown in Listing 6. First, the inlined state of the node n is set to false. The node’s tuple is initialized so that the cost is the IR size of n, and the benefit is the local benefit of n reduced by the local benefits of n’s children. The reasoning is that inlining n on its own forfeits the benefits of inlining its children, which would otherwise be realized if n was the compilation root.
The heuristic then sets the front of the node n to the set of its child clusters (note: at this point, the children of n are already assigned to clusters), to model the fact that n is initially alone in its cluster. As long as n’s front is not empty, the node with the largest (by ≥) tuple is selected. That tuple corresponds to the adjacent cluster with the largest benefit to cost ratio. The heuristic then checks whether inlining this adjacent cluster would increase the benefit to cost ratio of the current cluster. If yes, the clusters are merged by updating the tuple, front and inlined relations, and the loop repeats. If not, then there are no more clusters that could improve the current one, so the analysis of the node ends.

As an example, consider the cluster below the log method in Figure 3. If there is insufficient budget to inline the entire cluster, then log is compiled separately, in which case the length and the get methods remain direct calls on arrays, and only apply becomes polymorphic. This is much better than inlining the log method without the foreach due to insufficient budget, which leads to the foreach method getting compiled separately, making all of its callsites polymorphic.

Inlining. The bestCluster heuristic from Listing 5 uses the tuple value of a node n to select the cluster with the highest benefit to cost ratio. Once selected, the canInline heuristic uses the following threshold to decide whether to inline:

$$\langle \text{tuple}(n) \rangle \geq t_1 \cdot 2^{|\text{ir(root)}|+|\text{ir(n)}|}(16 - t_2)$$  \hspace{1cm} (12)

When the inlining starts and the root is small, the benefit to cost ratio may be small to justify inlining. However, as the inlining progresses, and the root method becomes larger, a callsite’s benefit must increasingly outweigh the method size. Importantly, the threshold is sensitive to the size of the method due to the $|\text{ir(n)}|$ term in the exponent, so it is "more forgiving" towards small methods when it gets close to the threshold. We therefore call this threshold function adaptive. We experimentally determined that the values $t_1 = 0.005$ and $t_2 = 120$ work well in our implementation.

For example, consider the println call from Figure 3. If the inliner is close to running out of budget, it might not make sense to inline a huge cluster, such as the subtree below log. However, exceeding the threshold only slightly to inline the println call, which is a bridge method for printf, does make sense because printf is a small method, and its main overhead is in the extra function call to the printf method.

Termination detection. We stop when there are no cutoff nodes left, i.e. $N_L(root) = 0$, or if there were no changes in the call tree during the last round. As a fallback, we also stop if the IR size of the root method exceeds 50000, since Graal’s compilations become too slow thereafter.

Deep inlining trials. By specializing the call nodes with their callsites’ arguments, our algorithm effectively simulates the optimizations that would later occur if the corresponding methods were inlined. The optimization count is used in the local benefit calculation in Equation 4. Furthermore, these optimizations simplify the call tree, thereby increasing the benefits and decreasing the costs of the call nodes. Such simplified call nodes are more likely to get inlined.

After inlining, we find all the callsites in the root method whose arguments were altered. For each such callsite, we copy the callsite arguments into the IR of the corresponding call node. We then propagate the improved type information through the IR, and trigger a Graal transformation called canonicalization. This phase includes a set of optimizations, such as constant folding [57], strength reduction [16], branch pruning, global value numbering [15], and JVM-specific simplifications such as type-check folding for values of known type. This process is repeated recursively in the call tree.

As an example, consider the method foreach from Figure 3. When main is the compilation root, propagating the callsite arguments allows devirtualizing and expanding the length, get and apply calls in the foreach. Without doing so, these calls remain polymorphic, which prevents further expansion, as well as concluding that foreach is beneficial.

Other optimizations. Additionally, we apply read-write elimination in the root method at the end of every round [81]. We found that this helps in some programs by partially restoring the method receiver type information that is lost when writing values to memory (and later reading the same values). At the end of every round, we also apply peeling on a loop’s first iteration if we detect that the loop contains a ϕ-node (i.e. a variable) whose type is more specific in that first iteration.

Polymorphic inlining. To inline polymorphic calls, we use the approach by Hölzle and Ungar [44]. We use a new call node kind P to model a polymorphic callsite. Each child node of a polymorphic callsite corresponds to a concrete target, where the targets are speculated based on the VM’s receiver type profile. When a polymorphic call node gets inlined, we emit a typeswitch (i.e. an if-cascade with type checks).

We experimentally found that a maximum of 3 targets, where each target must have at least a 10% probability, is usually a good tradeoff against the typeswitch overhead. If some types target the same method (e.g. due to subclassing), we check against the method’s address, as described by Detlefs and Agersen [19]. Depending on the profile, the typeswitch ends with a virtual call, or a deoptimization.
We use the following expression for benefit estimation:

\[ B_L(n) = \sum_{m \in \text{children}(n)} p_m \cdot B_L(m) \]  

(13)

Above, the value \( p_m \) is the profile-based probability that the virtual call dispatches to the respective child node \( m \).

**Recursive methods.** To prevent recursive calls from monopolizing the call graph exploration, we decrease the intrinsic exploration priority \( P_I(n) \) of a cutoff node \( n \) by a penalty \( \psi_r(n) \), where \( d(n) \) is the (possibly indirect) recursion depth.

\[ \psi_r(n) = \max(1, f(n)) \cdot \max(0, 2^{d(n)} - 2) \]  

(14)

The frequency \( f(n) \) is used to compensate the impact of the frequency multiplier in the local benefit estimation in Equation 4. This heuristic creates an increasing pressure against recursive methods. Until the recursion depth 2, the value of \( \psi_r \) is 0, but it increases exponentially thereafter.

**Parameter tuning.** All the parameters used in our implementation were tuned by doing an exhaustive search over ranges of values that we defined manually based on our intuition. In the tuning process, we selected the parameter configuration that resulted in the best peak performance across all benchmarks, but at the same time did not cause more than a 5% slowdown on any benchmark with respect to the existing inliners in Graal. Furthermore, another constraint was not to increase the warmup time by more than 20%. We did not optimize for the code size during the tuning, but our subsequent inspection revealed that the code size increase is not dramatic.

V. Evaluation

The goal of this evaluation is twofold. (1) We show that the adaptive inlining threshold in the expression in Equation 12 outperforms fixed thresholds, that clustering in Listing 6 outperforms method-by-method inlining, and that deep inlining trials from Section IV outperform normal inlining trials. To do this, we compare the tuned version of our algorithm against a range of possible parameters, (2) We show that our algorithm significantly outperforms existing inlining algorithms when applied to our compiler, as well as other JIT compilers for the JVM such as the default C2 compiler [46].

We conducted the experiments on an Intel i7-4930MX quad-core CPU with hyperthreading. We followed established practices for benchmarking on the JVM [26]. For each datapoint, we executed the measurements in 5 separate JVM instances, and we report both the mean and standard deviation. In each JVM instance, we measured peak performance – we repeated each benchmark a predefined number of times, and we computed the average of the last 40% (but at most 20) repetitions. Importantly, repetitions are chosen for each benchmark separately so that we always measure the steady state. The warmup curves for different alternatives reach stability after a similar time (i.e. after a similar number of repeated iterations of the benchmark) indicating that we tuned the inlining algorithm so that it does not incur a significant compilation overhead – we show only the most prominent examples in Figure 5.

To eliminate the effects of dynamic frequency scaling, we disabled the `intel_pstate` driver, and we set the frequency of all CPU cores to 3 GHz. We monitored the CPU frequency during benchmarking, and ensured that it stays at 3 GHz.

In the plots, we show both the running time, shown with curves, and the amount of code installed by the JIT compiler, superimposed with bars. We include the standard deviation for time, but not for code size, whose variance was very low.

We used 10 DaCapo benchmarks that run on JDK 8 [9]: all 12 Scala DaCapo benchmarks [78]: 3 benchmarks from the Spark Perf suite: the Gaussian mixture model, the decision tree and the multinomial naive Bayes algorithm for Apache Spark MLLib [64]: a set of Neo4J graph processing queries; a new Scala compiler implementation called Dotty; and the STMBench7 benchmark [28] applied to ScalaSTM [10].
Adaptive inlining threshold. We stress that, for us, running time is more important than code size. As argued before [80, 64], and as showed here, Graal’s optimizations are already very good. A 10% performance increase is regarded as a significant improvement, while a code size increase of up to 100% is typically tolerable (for example, GCC inliners report 44% binary size growth [65], and ≈100% increase is acceptable in some WCET-aware optimizations [22]). We therefore adjust the y-axis range to clearly show the area around the optimal values.

Adaptive inlining threshold. To show that the adaptive expansion threshold from Equation 8 and the inlining threshold from Equation 12 outperform fixed thresholds [6] [35], we implement an alternative policy that compares the call tree size to a fixed value $T_e$ to decide whether to continue expansion, and the root node count to $T_i$ to decide to continue inlining. We test $T_e \in \{500, 1k, 3k, 5k, 7k\}$, and $T_i \in \{1k, 3k, 6k\}$ (we found that benchmarks are most efficient in this range), and compare them against the tuned version of our algorithm. All other aspects of the algorithm are left as-is.

Figures 6 and 7 show that the fixed threshold can achieve the performance of the adaptive threshold heuristic, but the optimal parameters have to be tuned differently for every benchmark. For example, avrora and scalatest achieve the best performance for $T_e = 500$, but sunflow is 20% slower in that range compared to its optimal value $T_e = 1000$. On many benchmarks, such as fop, luindex, pmd, sunflow, kiana, scalac, scalariform, naive-bayes, dotty and stmbench?, the value $T_e = 1000$ seems like a good choice, but this value is ≈20% slower for jython, ≈5% slower for h2, ≈11% slower for factorie, ≈7% slower for scaladoc, ≈51% slower for gauss-mix. On the other hand, $T_i = 6000$ works well for jython, factorie and gauss-mix, but this value is an extremely bad choice for most other benchmarks.

Out of 28 benchmarks, the fixed threshold outperformed the adaptive threshold by more than 10% only in scalatest, and by about 5% in jython and dec-tree. However, the optimal parameters for scalatest are $T_e = 500$, $T_i = 1000$, for jython $T_e = 5k$, $T_i = 6k$, and for dec-tree $T_e = 1k$. The adaptive heuristic always outperformed the fixed one on luindex, actors, factorie, scaladoc, and gauss-mix.

On factorie, tm and gauss-mix, the optimal variant of fixed heuristic installed $2\times$ more code than adaptive, while the adaptive heuristic installed $2\times$ more code only on sunflow.

Clustering. To compare our clustering heuristic against the classic approach where each method’s benefit must exceed some threshold [6] [35] [22] [4], we implement a new analysis policy that assigns each method into a separate cluster. We compare the two heuristics, while leaving the rest of the algorithm as-is. For space reasons, we show some of the graphs in Figure 8 and we keep the rest in the appendix.

The 1-by-1 heuristic is quite sensitive to the parameters from Equation 12. In many benchmarks, $t_1 = 0.0001$ and $t_2 = 1440$ is the best choice (sunflow, xalan, and factorie, to name a few). However, this combination is slower by ≈12% for pmd, by ≈4% for apparat, by ≈8% for scaladoc, by ≈24% for scalaxb and by ≈8% for neo4j. By contrast, clustering is relatively insensitive to the choice of parameters, and either matches or outperforms the best 1-by-1 variant.
HotSpot C2 (server)

1-by-1 consistently outperforms clustering in Deep inlining trials. However, on Scala DaCapo, our inliner improves over C2 by \( \approx 8\% \) on pmd, \( \approx 13\% \) on factorie, \( \approx 7\% \) on scalariform, and \( \approx 6\% \) on scalatest. As for the other benchmarks, dotty is improved by \( \approx 2.5\% \), neo4j by \( \approx 6.5\% \), and gauss-mix is most affected with a \( \approx 59\% \) improvement.

Comparison against alternatives. To show the benefits of specializing the call tree by forwarding the callsite arguments and triggering the optimizations, we compare them against the approach of specializing the callsites only in the root compilation method \( \approx 6\% \). The results are shown in Figure 9 where the proposed inliner with deep trials is compared against the same inliner implementation in Graal that does not do deep trials (blue and the green bars).

Overall, deep inlining trials have very small impact on DaCapo benchmarks. However, on Scala DaCapo, the improvement is \( \approx 8\% \) on actors, \( \approx 13\% \) on factorie, \( \approx 7\% \) on scalar, and \( \approx 6\% \) on scalatest. As for the other benchmarks, dotty is improved by \( \approx 2.5\% \), neo4j by \( \approx 6.5\% \), and gauss-mix is most affected with a \( \approx 59\% \) improvement.

Comparison against alternatives. To show the benefits of the proposed inliner, we compared our implementation against the inliner implementation that is available in the open-source Graal, which is available on GitHub \([3]\). This inliner is akin to the inlining algorithm for JIT compilers described by Steiner et al. \([82]\), which does not have an exploration phase. We stress that we used the Enterprise Graal for the comparison – the only component that we replaced was the inliner. We also compared against the standard HotSpot C2 compiler, which inlines a single-method at a time (first only trivial methods during bytecode parsing, and larger methods in a separate, later phase), with a greedy heuristic that is similar to the one used in basic Graal. The results are shown in Figure 9.

On all benchmarks except pmd, our algorithm outperforms Graal’s open-source inliner, in some cases by several times. With respect to C2 and DaCapo, we observed improvements of \( \approx 21\% \) on jython, \( \approx 13\% \) on luindex, \( \approx 5.5\% \) on pmd, and \( \approx 9\% \) on sunflow. C2 outperforms Graal on lusearch and xalan. We note that C2 was heavily optimized for DaCapo.

On Scala DaCapo, our inliner improves over C2 by \( \approx 1.7\times \) on apparat, \( \approx 2.9\times \) on factorie, \( \approx 1.45\times \) on kiama, \( \approx 1.45\times \) on scaladoc, \( \approx 1.5\times \) on tnt, \( \approx 1.8\times \) on naive-bayes, and \( \approx 1.9\times \) on gauss-mix. C2 outperforms our inliner by around 10% on scalatest and by 4% on neo4j.
without any second-tier compiler – since the first-tier compiler compiles only methods that are hot. We found that the C1 code second-tier compiler is usually not critical, since the second-tier machine code, but Graal generates substantially faster code. In some cases, Enterprise Graal with the proposed inliner produces more code than the alternatives. However, this is not always the case – by applying other optimizations to simplify the call graph, the proposed inliner was able to make more callsites direct, which enabled additional inlining.

VI. RELATED WORK

Aside from decreasing the cost of method calls, the main benefit of inlining is, for many compilers, that it enables additional optimizations. Thus, inlining eliminates abstractions in the program with the aim of making it more low-level. This includes traditional abstractions such as lambdas [24], bulk collection processing [61, 48, 21, 83, 69, 72], various object-oriented design patterns [25], functional programming patterns [43, 86], data structures [55, 56, 57, 52, 71, 62, 59, 60, 49], and event streams [45, 63], but also async/await [29], futures and promises [31], message-passing models [70, 58, 30, 51, 53, 50], dataflow concurrency [68, 75, 67], and coroutines [66, 63].

In this section, we survey the related work on inline-substitution algorithms, and other compiler optimizations that are related to the proposed algorithm [54].

Priority-guided inlining. Ayers et al. described an inlining algorithm that traverses the callsites in the program according to their priority [6], and a similar algorithm was later studied by Arnold et al. [5]. A variant of this algorithm was used in GCC [35], and in the context of C compilers for embedded systems [4]. In their budget-driven priority-based inlining, callsite priority is equal to the estimated benefit of inlining the call, and inlining stops once the total cost of the inlined method exceeds a fixed threshold. Zhao and Amaral showed that using benefit divided by cost improves the inlining decisions, and they used the total application size metric to scale the cost of small methods [92].

Steiner et al. compared depth-first, breadth-first and priority inlining in a JIT compiler, and reported minor differences

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Graal, new</th>
<th>Graal, greedy</th>
<th>HotSpot C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>avora</td>
<td>2.39</td>
<td>1.17</td>
<td>1.25</td>
</tr>
<tr>
<td>bank</td>
<td>8.35</td>
<td>3.60</td>
<td>4.39</td>
</tr>
<tr>
<td>jython</td>
<td>6.93</td>
<td>3.81</td>
<td>4.70</td>
</tr>
<tr>
<td>luindex</td>
<td>17.00</td>
<td>3.46</td>
<td>4.95</td>
</tr>
<tr>
<td>lusearch</td>
<td>24.34</td>
<td>14.86</td>
<td>11.30</td>
</tr>
<tr>
<td>pinl</td>
<td>1.96</td>
<td>1.06</td>
<td>1.43</td>
</tr>
<tr>
<td>pinl</td>
<td>14.39</td>
<td>5.79</td>
<td>6.72</td>
</tr>
<tr>
<td>snowfall</td>
<td>1.91</td>
<td>1.18</td>
<td>0.96</td>
</tr>
<tr>
<td>xalan</td>
<td>5.78</td>
<td>2.39</td>
<td>3.71</td>
</tr>
<tr>
<td>actors</td>
<td>6.57</td>
<td>2.01</td>
<td>2.41</td>
</tr>
<tr>
<td>apprent</td>
<td>8.46</td>
<td>3.13</td>
<td>4.07</td>
</tr>
<tr>
<td>factorie</td>
<td>3.32</td>
<td>1.36</td>
<td>1.17</td>
</tr>
<tr>
<td>kiama</td>
<td>4.61</td>
<td>2.29</td>
<td>3.14</td>
</tr>
<tr>
<td>scalac</td>
<td>28.40</td>
<td>15.43</td>
<td>25.21</td>
</tr>
<tr>
<td>scaladoc</td>
<td>16.16</td>
<td>9.16</td>
<td>13.34</td>
</tr>
<tr>
<td>scalap</td>
<td>4.52</td>
<td>1.61</td>
<td>1.82</td>
</tr>
<tr>
<td>scalarform</td>
<td>8.69</td>
<td>3.31</td>
<td>4.73</td>
</tr>
<tr>
<td>scalatest</td>
<td>12.31</td>
<td>6.06</td>
<td>12.42</td>
</tr>
<tr>
<td>scalacb</td>
<td>8.81</td>
<td>2.99</td>
<td>4.91</td>
</tr>
<tr>
<td>specs</td>
<td>8.46</td>
<td>3.20</td>
<td>4.20</td>
</tr>
<tr>
<td>tnt</td>
<td>5.22</td>
<td>1.65</td>
<td>2.13</td>
</tr>
<tr>
<td>naive-bayes</td>
<td>3.73</td>
<td>2.35</td>
<td>2.72</td>
</tr>
<tr>
<td>dec-tree</td>
<td>15.25</td>
<td>6.33</td>
<td>8.51</td>
</tr>
<tr>
<td>gaua-mix</td>
<td>8.53</td>
<td>4.12</td>
<td>5.36</td>
</tr>
<tr>
<td>donut</td>
<td>25.56</td>
<td>15.02</td>
<td>22.50</td>
</tr>
<tr>
<td>newslq</td>
<td>28.82</td>
<td>12.60</td>
<td>17.37</td>
</tr>
<tr>
<td>snibench7</td>
<td>6.68</td>
<td>1.95</td>
<td>2.19</td>
</tr>
</tbody>
</table>

Code size comparison. In Figure 10 we compare the size of the machine code produced by Enterprise Graal with the proposed inlining algorithm against the code size of HotSpot C2, HotSpot C1, and open-source Graal. We only include the DaCapo and Scala DaCapo benchmarks on which we observed the most prominent differences in Figure 9. In most cases, Enterprise Graal with the proposed inliner produces more code than the alternatives. However, this is not always the case – for example, on luindex, scalac and scaladoc, Graal and C2 produce approximately the same amount of machine code, but Graal generates substantially faster code. Interestingly, on lusearch, where C2 outperformed Graal, the amount of generated machine code is similar. We also included the code size for the first-tier C1 compiler when it is used without any second-tier compiler – since the first-tier compiler anyway compiles more methods (shown with the transparent bars in Figure 10), this illustrates that the total code size of the second-tier compiler is usually not critical, since the second-tier compiles only methods that are hot. We found that the C1 code size generally varies a lot (depending on the benchmark), and is in some cases higher than Graal EE.

In Table 1 we show a more algorithm-focused comparison between total size of the code generated by Graal with our new inliner, Graal with the greedy inliner, and HotSpot C2. We found that Graal with the proposed inlining algorithm on average generates ≈1.88× more code than C2, and on average ≈2.37× more code than Graal with the greedy inliner. We note that the tuning the inlining parameters of the greedy inliner to do more inlining did not substantially improve performance, and in some cases did not even increase the code size. We found that the increase in the inlining budget does not necessarily enable more inlining – by applying other optimizations to simplify the call graph, the proposed inliner was able to make more callsites direct, which enabled additional inlining.
between these approaches [82]. Their implementation did not alternate between exploration and inlining – we found that this substantially improves performance.

Priority-driven inlining inspired our own algorithm design, but we found that jointly inlining clusters of callsites instead of single callsites, and using an adaptive inlining threshold instead of a fixed threshold, substantially improves priority-based inlining.

**Prediction-driven inlining.** Multiple inlining heuristics based on a fixed set of decision rules were proposed [76, 47, 35]. Cooper et al. showed experimental evidence that program-specific inlining heuristics outperform one-size-fits-all inlining heuristics [17]. In particular, some heuristics prefer callsites whose inlining later leads to better optimizations. The idea of predicting the optimizations enabled by inlining was first proposed by Ball [7], and later explored by Dean and Chambers in the context of Self [13]. In both cases, the inlining benefits were predicted only one level below the root compilation method. Waddell and Dybvig showed that alternating inlining with optimizations such as constant folding produces more efficient code [85]. Jagannathan and Wright showed that a dataflow analysis on Scheme programs allows identifying callsites that are transfigually called with more specific types [36]. Similarly, Sewe et al. showed that propagating the argument type information allows more easily identifying optimizations, and making better inlining decisions [77]. Our algorithm both propagates argument types, and recursively executes optimizations to specialize the call tree. This allows measuring optimization impact deep inside the call tree in a context-sensitive way.

The inliner in the LLVM infrastructure [39] traverses the strongly connected components of the call-graph bottom-up, inlining methods one-by-one [2]. This inliner uses two fixed inlining thresholds depending on whether the method is regarded as hot [91, 90]. Benefit is estimated using a one-level inlining trial [7, 18, 11].

**Profile-driven inlining.** Profile-based inlining was studied by Scheifler in the context of CLU [74]. Chang et al. showed that the execution profiles can improve inlining in C programs [12, 13], and McFarling described a technique that minimizes the instruction cache miss rate [44]. Our inlining algorithm relies on branch profiles to calculate the callsite frequencies, and it additionally uses type profile information to speculative inline callsites that are polymorphic.

**Other techniques.** In the context of trace-based JIT compilers, Haeubl et al. studied trace inlining for Java [32], can result in good performance. Our implementation does not target a trace-based JIT compiler, and we did not yet investigate trace-based inlining in detail. Hazelwood and Grove found that guiding the inlining decisions with context-sensitive profiles improves performance [33]. We believe that context-sensitive profiles could improve our algorithm further, but we did not yet evaluate this, since the profiles provided by the HotSpot VM are context-insensitive. Some researchers have also observed the impedance between methods as logical units of functionality, and the optimizable regions in the program, which drove the procedure-boundary-elimination technique for whole-program optimization [84]. Unlike our work, where we inline a method once we realize its optimization potential, and then execute the optimizations, procedure boundary elimination transfers knowledge between compilation units, and can apply some optimizations without inlining. Other related techniques, such as procedure strength reduction and procedure vectorization, were done in the context of telescoping languages [14].

Simon et al. used machine learning to synthesize inlining heuristics, and reported performance improvements between 3% and 9% [79]. Cammarota et al. also investigated how to use machine learning to select between inlining heuristics, in the context of the GCC compiler [11]. We did not investigate if machine learning can improve the decision points in our inlining algorithm, and we leave that to future work.

Polymorphic inlining was studied by Hölzle and Ungar [34], while Detlefs and Agesen proposed a variant that compares method addresses instead of receiver types [19].


Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.


Scott McFarling. 1991. Procedure Merging with In-


