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, STMBench7 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 $3\times$.

***Index Terms*—just-in-time compilation, inlining, polymorphic dispatch, cost-benefit analysis, optimization-driven inlining, priority-based inlining, inline subtitution, 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, 5]. *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 optimizable 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[1] [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 [3], and we report performance

[1] Artifact available at: https://doi.org/10.5281/zenodo.2328430

improvements ranging from 5% up to $3\times$. 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 callsites at any given point in time.

Online inlining problem. Given a set Π of program methods M_i, 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 Π, 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 callsite C_j to a method M is inlined, runtimes such as the JVM stop measuring the hotness of M at the callsite 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 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.

```
def main(args: Array[String]) {
  async { log(getStackTrace) }
  log(args)
}
def log[T](xs: Array[T]) {
  xs.foreach(println)
}
trait IndexedSeqOptimized[T] {
  def get(i: Int): T
  def length: Int
  def foreach(f: T => Unit) {
    var i = 0
    while (i < this.length)
      { f.apply(this.get(i)); i += 1 }
  }
}
```

Fig. 1: An Example Scala Program

Listing 1: Incremental Inlining Algorithm

```
input  : root compilation method $\mu$
output : method $\mu$ with inlined callsites
root = createRoot($\mu$);
while $\neg$ detectTermination(root) do
    expand(root);
    analyze(root);
    inline(root);
end
```

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 IV. 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 6) 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 μ, called the *root compilation*

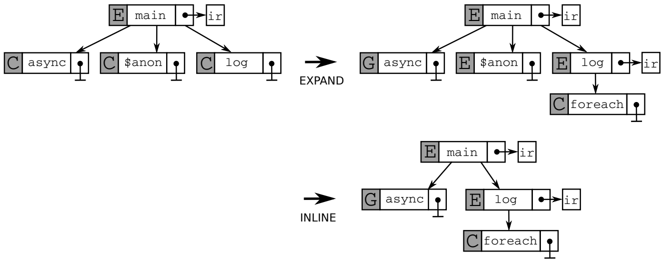


Fig. 2: Call Tree During an Inlining Round

Listing 2: Call Tree Node

```
Struct Node is
    kind: one of { E, C, G, D };
    ir: corresponding method body;
    callsite: pointer to the callsite in the parent;
    children: list of pointers to the children;
end
```

method. The algorithm creates the `root` call tree node for μ, which contains its intermediate representation and the list of *child nodes*. Each child corresponds to a callsite in μ. 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 μ. 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 these 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, 82], 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 μ and a connected subset of the nodes reachable from μ.

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, 92], 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 [27, 23].
(3) *Callsite specialization*: A callsite within the root method μ can be specialized with the callsite arguments in μ'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.

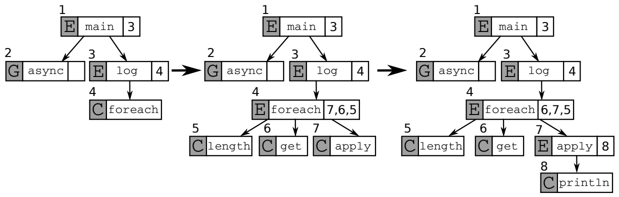
Description. The `expand` procedure in Listing 3 initializes a mapping `queue` between each node and the set of its children that should be considered for expansion. The `expand` then repeatedly descends into the call tree, until the `expansionDone` heuristic interrupts it. The `descend` procedure selects 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.

Listing 3: Expansion Phase

```
input   : call tree $root$
output  : expanded call tree
Procedure expand(root) is
    for $n \in root$ do queue(n) = n.children;
    while $\neg$ expansionDone(root) do descend(root);
end
Procedure descend(n) is
    if n.kind $\in$ { E } then
        best = highestPriorityNode(queue(n));
        queue(n) = queue(n) $\setminus$ best;
        new = descend(best);
        if new $\neq$ null $\wedge$ |queue(new)| > 0 then
            queue(n) = queue(n) $\cup$ new;
        end
        return n;
    else if n.kind $\in$ { C } then return expandCutoff(n);
end
```

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

Listing 4: Cost-Benefit Analysis Phase

```
input   : call tree without cost-benefit decisions
output  : call tree with cost-benefit decisions
Procedure analyze(node) is
    foreach c $\in$ node.children do analyze(c);
    analyzeNode(node);
end
```

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.

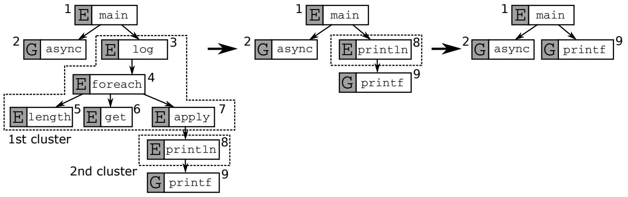


Fig. 3: Analysis and the Inlining Phase Example

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 reduction in the program runtime) and the cost (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

Listing 5: Inlining Phase

```
input   : root of the analyzed call tree
output  : root with inlined callsites
Procedure inline(root) is
    queue = $\varnothing$;
    foreach c $\in$ root.children do queue = queue $\cup$ c;
    while $\neg$ isEmpty(queue) do
        n = bestCluster(queue);
        queue = queue $\setminus$ n;
        if canInline(root, n) then inlineCluster(n, queue);
    end
end
Procedure inlineCluster(n, queue) is
    inlineIR(n.callsite, n.ir);
    foreach c $\in$ n.children do
        root.children = root.children $\cup$ c;
        if inCluster(n, c) then inlineCluster(c, queue);
        else queue = queue $\cup$ c;
    end
end
```

Description. 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 3 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 [46, 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 $S_{ir}(n)$ of *all* the call node IR sizes in that subtree.

$$S_{ir}(n) = \sum_{m \in subtree(n)} |ir(m)| \tag{1}$$

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

$$S_b(n) = \sum_{m \in subtree(n), kind(m)=C} |ir(m)| \tag{2}$$

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

$$N_c(n) = |m \in subtree(n) : kind(m) = C| \tag{3}$$

For a particular callsite n with a frequency $f(n)$ relative to the root, let $N_s(n)$ be the number of arguments whose type is more concrete than the formal parameters, and $N_o(n)$ 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.

$$B_L(n) = \begin{cases} f(n) \cdot (1 + N_s(n)) & kind(n) = C \\ f(n) \cdot (1 + N_o(n)) & kind(n) = E \end{cases} \tag{4}$$

Note that, in our implementation, we calculate $N_o(n)$ 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* $P_I(n)$ for a node n as:

$$P_I(n) = \begin{cases} \frac{B_L(n)}{|ir(n)|} & kind(n) = C \\ \max\limits_{c \in \{n\} \cup children(n)} P_I(c) & kind(n) = E \end{cases} \tag{5}$$

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 $P_I(n)$ decreased by the penalty $\psi(n)$:

$$P(n) = P_I(n) - \psi(n) \tag{6}$$

We use $\psi(n)$ 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).

$$\psi(n) = p_1 \cdot S_{ir}(n) + p_2 \cdot S_b(n) - b_1 \cdot \max(0, b_2 - N_c^2(n)) \tag{7}$$

The penalty ψ correlates with S_{ir} and S_b, 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 those few cutoffs would turn the entire subtree into a single cluster. We experimentally tuned Graal to use the values $p_1 = 10^{-3}$, $p_2 = 10^{-4}$, $b_1 = 0.5$, and $b_2 = 10$, 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:

$$\frac{B_L(n)}{|ir(n)|} \geq e^{(S_{ir}(\texttt{root}) - r_1)/r_2} \tag{8}$$

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 $r_1 \approx 3000$ and $r_2 \approx 500$ 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 $\oplus$:

$$b_1|c_1 \oplus b_2|c_2 \equiv b_1 + b_2|c_1 + c_2 \tag{9}$$

The second tuple operation is comparison $\circledgeq$:

$$b_1|c_1 \circledgeq b_2|c_2 \Leftrightarrow \frac{b_1}{c_1} \geq \frac{b_2}{c_2} \tag{10}$$

Finally, we define the cost to benefit ratio $\langle . \rangle$ as follows:

$$\langle b|c \rangle \equiv {}^{b}\!/_{c} \tag{11}$$

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.

Listing 6: Cost-Benefit Analysis Phase

```
Procedure analyzeNode(n) is
    inlined(n) = false;
    tuple(n) = $B_L(n) - \sum_{m \in children(n)} B_L(m) \Big| |ir(n)|$;
    front(n) = n.children;
    while front(n) $\neq \varnothing$ do
        m = $\arg\max_{d \in front(n)} tuple(d)$;
        if tuple(n) $\oplus$ tuple(m) ⧁ tuple(n) then
            tuple(n) = tuple(n) $\oplus$ tuple(m);
            front(n) = front(n) $\setminus$ m $\cup$ front(m);
            inlined(m) = true
        end
        else break;
    end
end
```

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:

$$\langle tuple(n) \rangle \geq t_1 \cdot 2^{(|ir(root)|+|ir(n)|)/(16 \cdot t_2)} \qquad (12)$$

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 $|ir(n)|$ 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 `println` 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. $N_c(root) = 0$, 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 [87], 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 [34]. 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 trade-off 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 Agesen [19]. Depending on the profile, the typeswitch ends with a virtual call, or a deoptimization.

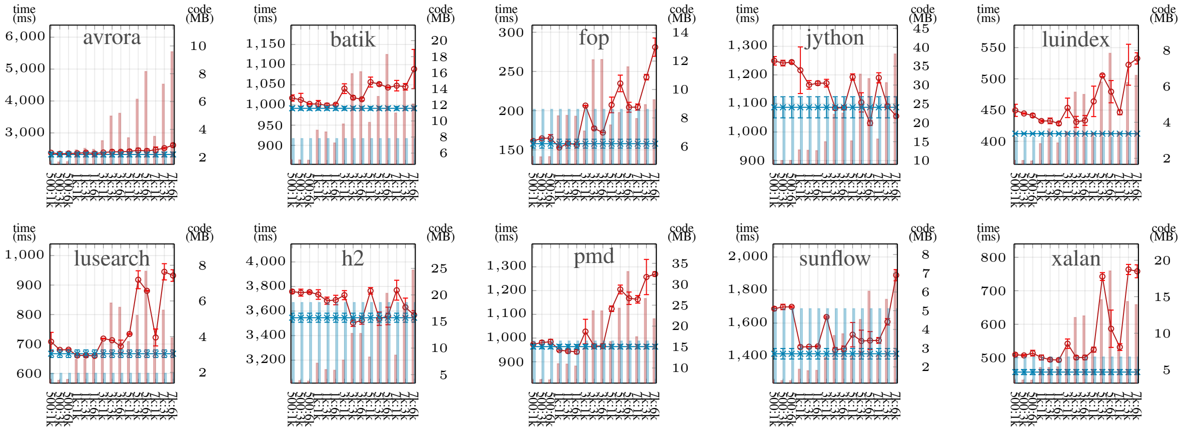


Fig. 4: DaCapo: Fixed (red ○) vs Adaptive (blue ×) Threshold (bars – memory, curves – running time, x-axis – T_e;T_i)

We use the following expression for benefit estimation:

$$B_L(n) = \sum_{m \in children(n)} p_m \cdot B_L(m) \tag{13}$$

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.

$$\psi_r(n) = \max(1, f(n)) \cdot \max(0, 2^{d(n)} - 2) \tag{14}$$

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 ψ_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].

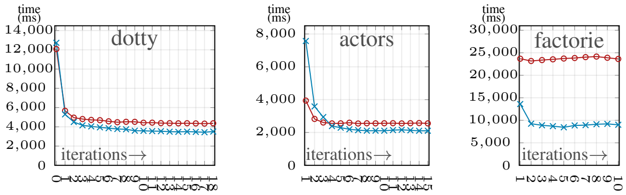


Fig. 5: Warmup Curves: C2 (red ○) vs Graal (blue ×)

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].

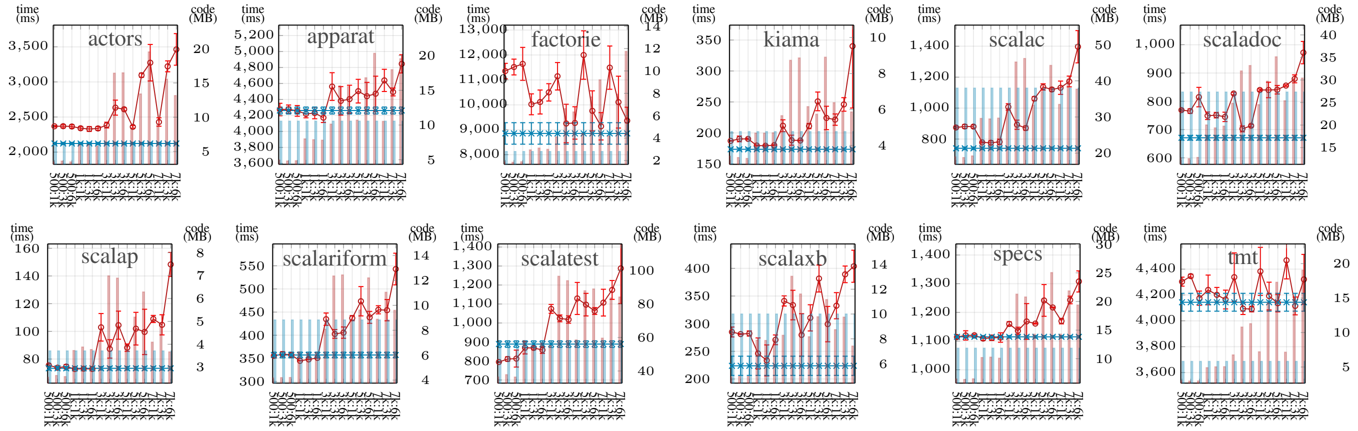


Fig. 6: Scala DaCapo: Fixed (red ○) vs Adaptive (blue ×) Threshold (bars – memory, curves – running time, x-axis – T_e;T_i)

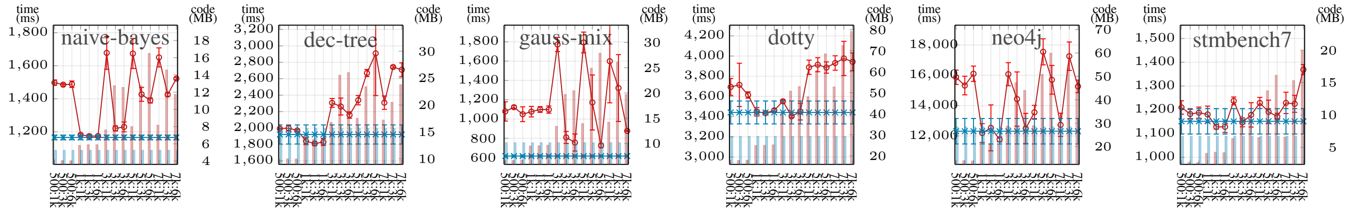


Fig. 7: Other: Fixed (red ○) vs Adaptive (blue ×) Threshold (bars – memory, curves – running time, x-axis – T_e;T_i)

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 [35], and ≈ 100% increase is acceptable in some WCET-aware optimizations [42]). 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, 5, 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 4, 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`, `kiama`, `scalac`, `scalariform`, `naive-bayes`, `dotty` and `stmbench7`, 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`, `tmt` and `gauss-mix`, the optimal variant of fixed heuristic installed 2× more code than adaptive, while the adaptive heuristic installed 2× 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, 5, 92, 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 `scalatest`, 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.

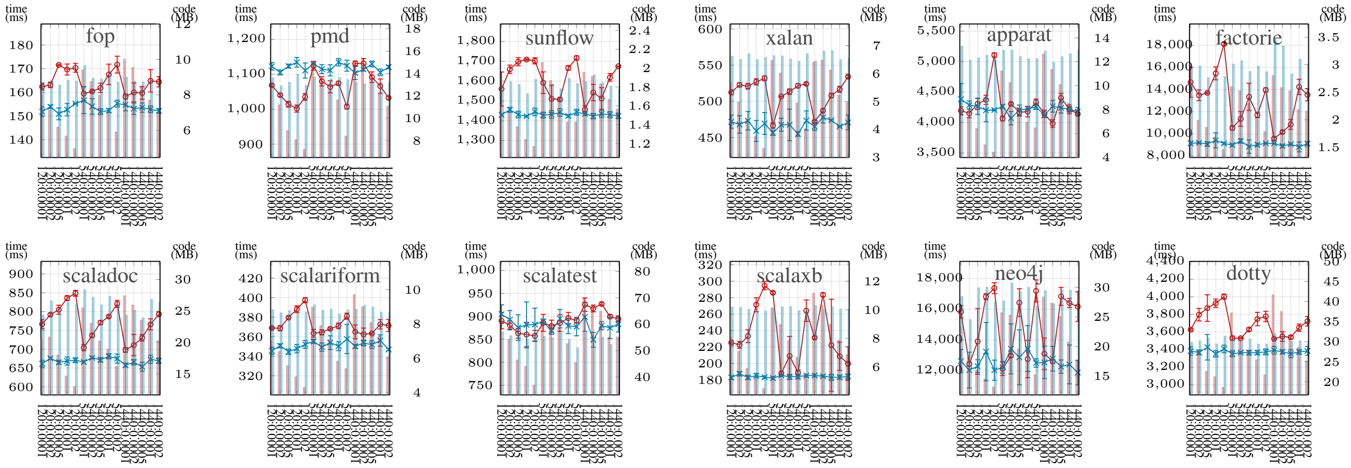


Fig. 8: 1-By-1 (red ○) vs Clustering (blue ×) Decisions (bars – memory, curves – running time, x-axis – $t_2;t_1$)

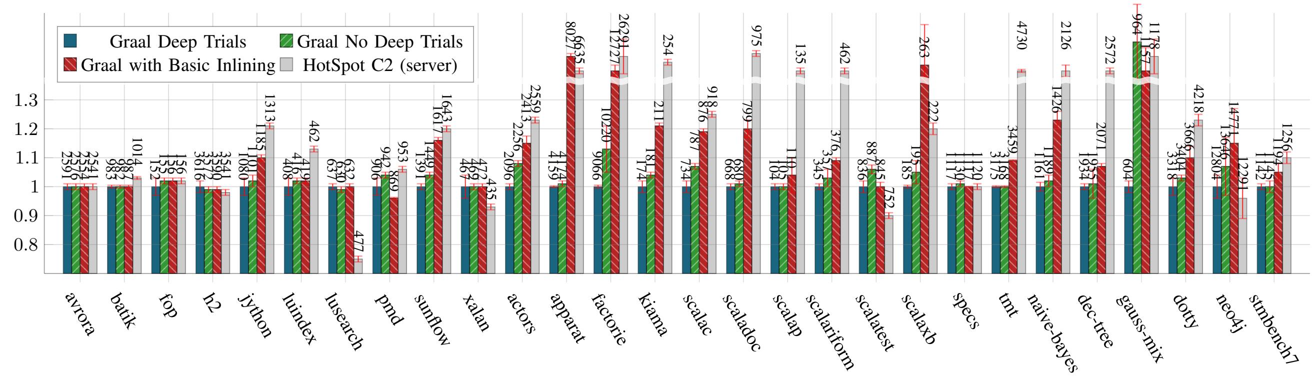


Fig. 9: Comparison of running times for different inlining algorithms and compilers (normalized, lower is better)

This time, clustering outperforms the 1-by-1 heuristic for all parameter combinations on `fop`, `factorie`, `scaladoc`, `scalariform` and `dotty`. The only benchmark on which 1-by-1 consistently outperforms clustering is `pmd`.

Deep inlining trials. 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 [7, 18]. 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 ≈ 8% on `actors`, ≈ 13% on `factorie`, ≈ 7% on `scalac`, and ≈ 6% on `scalatest`. As for the other benchmarks, `dotty` is improved by ≈ 2.5%, `neo4j` by ≈ 6.5%, and `gauss-mix` is most affected with a ≈ 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 respected to C2 and DaCapo, we observed improvements of ≈ 21% on `jython`, ≈ 13% on `luindex`, ≈ 5.5% on `pmd`, and ≈ 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 ≈ 1.7× on `apparat`, ≈ 2.9× on `factorie`, ≈ 1.45× on `kiama`, ≈ 1.45× on `scaladoc`, ≈ 1.5× on `tmt`, ≈ 1.8× on `naive-bayes`, and ≈ 1.9× on `gauss-mix`. C2 outperforms our inliner by around 10% on `scalatest` and by 4% on `neo4j`.

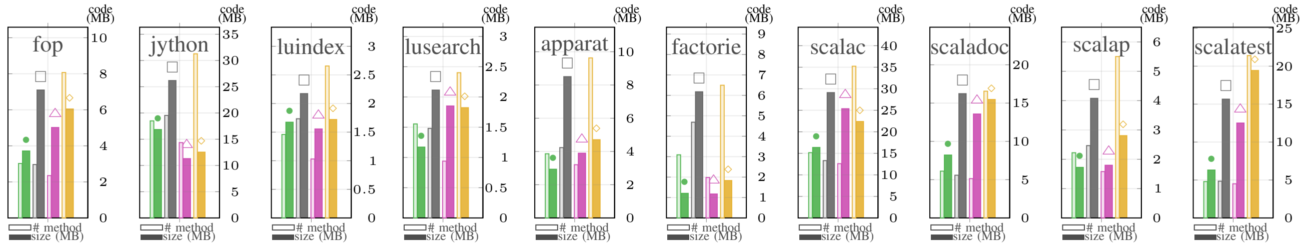


Fig. 10: Code size comparison between compilers (Graal CE ●, Graal EE □, C2 △, C1 ◇)

TABLE I: Size of the Generated Code (MB)

Benchmark	Graal, new	Graal, greedy	HotSpot C2
avrora	2.39	1.17	1.28
batik	8.35	3.60	4.89
fop	6.93	3.81	4.70
h2	17.00	3.46	4.95
jython	24.34	14.86	11.30
luindex	2.81	1.41	1.59
lusearch	1.96	1.06	1.43
pmd	14.39	5.71	6.72
sunflow	1.91	1.18	0.96
xalan	5.78	2.49	3.71
actors	6.57	2.01	2.41
apparat	8.46	3.13	4.07
factorie	3.32	1.36	1.17
kiama	4.61	2.29	3.14
scalac	28.40	15.43	25.21
scaladoc	16.16	9.16	13.34
scalap	4.32	1.61	1.82
scalariform	8.09	3.31	4.73
scalatest	12.31	6.06	12.42
scalaxb	8.61	2.99	4.91
specs	8.46	3.20	4.20
tmt	5.22	1.65	2.13
naive-bayes	3.73	2.35	2.72
dec-tree	15.35	6.33	8.51
gauss-mix	8.53	4.12	5.36
dotty	25.56	15.02	22.50
neo4j	28.82	12.60	17.37
stmbench7	6.68	1.95	2.19

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 I, 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.

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, 65].

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 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 [18]. 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 transitively 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, 1].

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].

VII. Conclusion

We presented a novel inlining algorithm for JIT compilers that makes use of three new heuristics – callsite clustering, adaptive thresholds and deep inlining trials. We experimentally showed that each of these heuristics improves the inlining algorithm on the standard benchmarks, without a serious increase in code size. In fact, in many benchmarks, we showed that excess inlining can be harmful and heuristics produce better results with a smaller code size. The evaluation showed that our new inlining algorithm significantly improves the existing state-of-the-art, such as the greedy inlining algorithm by Steiner et al. [82], and the inliner used by the HotSpot's C2 compiler. Speedups range from 5% up to $3\times$ on 21 out of 28 benchmarks, and only 4 benchmarks exhibit a slowdown of $5 - 20\%$ compared to the C2 compiler.

Even though this work establishes the importance of alternating graph exploration with optimizations and inlining, as well as the benefits of callsite clustering and adaptive thresholds, it also leaves several open research questions. For example, could better benefit estimation heuristics, based on more accurate cost models, further improve the proposed inlining algorithm? Could a dataflow-driven benefit estimation improve the performance of the generated code? Also, our algorithm used several carefully tuned parameters – can such parameters be tuned online using machine learning? We leave these and other questions to future work.

References

[1] 2018. LLVM Cost-Benefit Estimation Implementation at GitHub. (2018). https://github.com/llvm-mirror/llvm/blob/88ab6705571782fa664ecfa71b2f959a0daf2d78/lib/Analysis/InlineCost.cpp

[2] 2018. LLVM Inliner Implementation at GitHub. (2018). https://github.com/llvm-mirror/llvm/blob/6f1d64eb934e12ca5e8dcd378f88d1e6b80e8c55/lib/Transforms/IPO/Inliner.cpp

[3] 2018. Open-source Graal Repository at GitHub. (2018). http://github.com/graalvm/graal

[4] Pär Andersson. 2009. Evaluation of Inlining Heuristics in Industrial Strength Compilers for Embedded Systems. (2009).

[5] 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, USA, 52–64. https://doi.org/10.1145/351397.351416

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

[7] J. Eugene Ball. 1979. Predicting the Effects of Optimization on a Procedure Body. In *Proceedings of the 1979 SIGPLAN Symposium on Compiler Construction (SIGPLAN '79)*. ACM, New York, NY, USA, 214–220. https://doi.org/10.1145/800229.806972

[8] 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 Heidelberg, Berlin, Heidelberg, 384–397. https://doi.org/10.1007/978-3-540-77560-7_26

[9] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[10] Nathan Bronson, Jonas Boner, Guy Korland, Aleksandar Prokopec, Krishna Sankar, Daniel Spiewak, and Peter Veentjer. 2011. ScalaSTM Expert Group. (2011). https://nbronson.github.io/scala-stm/expert_group.html.

[11] Rosario Cammarota, Alexandru Nicolau, Alexander V. Veidenbaum, Arun Kejariwal, Debora Donato, and Mukund Madhugiri. 2013. On the Determination of Inlining Vectors for Program Optimization. In *Proceedings of the 22Nd International Conference on Compiler Construction (CC'13)*. Springer-Verlag, Berlin, Heidelberg, 164–183. https://doi.org/10.1007/978-3-642-37051-9_9

[12] P. P. Chang and W.-W. Hwu. 1989. Inline Function Expansion for Compiling C Programs. *SIGPLAN Not.* 24, 7 (June 1989), 246–257. https://doi.org/10.1145/74818.74840

[13] 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. https://doi.org/10.1002/spe.4380220502

[14] Arun Chauhan and Ken Kennedy. 2001. Optimizing Strategies for Telescoping Languages: Procedure Strength Reduction and Procedure Vectorization. In *Proceedings of the 15th International Conference on Supercomputing (ICS '01)*. ACM, New York, NY, USA, 92–101. https://doi.org/10.1145/377792.377812

[15] Cliff Click. 1995. Global Code Motion/Global Value Numbering. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI '95)*. ACM, New York, NY, USA, 246–257. https://doi.org/10.1145/207110.207154

[16] John Cocke and Ken Kennedy. 1977. An Algorithm for Reduction of Operator Strength. *Commun. ACM* 20, 11 (Nov. 1977), 850–856. https://doi.org/10.1145/359863.359888

[17] 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, Heidelberg, 69–84. http://dl.acm.org/citation.cfm?id=1788374.1788381

[18] Jeffrey Dean and Craig Chambers. 1994. Towards Better Inlining Decisions Using Inlining Trials. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming (LFP '94)*. ACM, New York, NY, USA, 273–282. https://doi.org/10.1145/182409.182489

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

[20] 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, USA, 1–10. https://doi.org/10.1145/2542142.2542143

[21] Michael Duigou. 2011. Java Enhancement Proposal 107: Bulk Data Operations for Collections. (2011). http://openjdk.java.net/jeps/107.

[22] 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 2017)*. ACM, New York, NY, USA, 92–104. https://doi.org/10.1145/3132190.3132209

[23] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. 2013. Efficient construction of approximate call graphs for JavaScript IDE services. In *2013 35th*

International Conference on Software Engineering (ICSE). 752–761. https://doi.org/10.1109/ICSE.2013.6606621

[24] Rémi Forax, Vladimir Zakharov, Kevin Bourrillion, Dan Heidinga, Srikanth Sankaran, Andrey Breslav, Doug Lea, Bob Lee, Brian Goetz, Daniel Smith, Samuel Pullara, and David Lloyd. 2014. Java Specification Request 335: Lambda Expressions for the JavaTM Programming Language. (2014). https://jcp.org/en/jsr/detail?id=335.

[25] 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.

[26] 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 and Applications (OOPSLA '07)*. ACM, New York, NY, USA, 57–76. https://doi.org/10.1145/1297027.1297033

[27] David Grove and Craig Chambers. 2001. A Framework for Call Graph Construction Algorithms. *ACM Trans. Program. Lang. Syst.* 23, 6 (Nov. 2001), 685–746. https://doi.org/10.1145/506315.506316

[28] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. 2007. STMBench7: A Benchmark for Software Transactional Memory. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys '07)*. ACM, New York, NY, USA, 315–324. https://doi.org/10.1145/1272996.1273029

[29] Philipp Haller, Simon Geries, Michael Eichberg, and Guido Salvaneschi. 2016. Reactive Async: Expressive Deterministic Concurrency. In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala (SCALA 2016)*. ACM, New York, NY, USA, 11–20. https://doi.org/10.1145/2998392.2998396

[30] Philipp Haller and Martin Odersky. 2007. Actors That Unify Threads and Events. In *Proceedings of the 9th International Conference on Coordination Models and Languages (COORDINATION'07)*. Springer-Verlag, Berlin, Heidelberg, 171–190. http://dl.acm.org/citation.cfm?id=1764606.1764620

[31] Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland Kuhn, and Vojin Jovanovic. 2012. Scala Improvement Proposal: Futures and Promises (SIP-14). http://docs.scala-lang.org/sips/pending/futures-promises.html

[32] Christian Häubl, Christian Wimmer, and Hanspeter Mössenböck. 2013. Context-sensitive Trace Inlining for Java. *Comput. Lang. Syst. Struct.* 39, 4 (Dec. 2013), 123–141. https://doi.org/10.1016/j.cl.2013.04.002

[33] Kim Hazelwood and David Grove. 2003. Adaptive Online Context-sensitive Inlining. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '03)*. IEEE Computer Society, Washington, DC, USA, 253–264. http://dl.acm.org/citation.cfm?id=776261.776289

[34] Urs Hölzle and David Ungar. 1994. Optimizing Dynamically-dispatched Calls with Run-time Type Feedback. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94)*. ACM, New York, NY, USA, 326–336. https://doi.org/10.1145/178243.178478

[35] Jan Hubicka. 2004. The GCC call graph module: a framework for inter-procedural optimization. In *Proceedings of the 2004 GCC Developers' Summit*. 65–78.

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

[37] Michael R. Jantz and Prasad A. Kulkarni. 2013. Performance Potential of Optimization Phase Selection During Dynamic JIT Compilation. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '13)*. ACM, New York, NY, USA, 131–142. https://doi.org/10.1145/2451512.2451539

[38] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. 2008. Design of the Java HotSpot&Trade; Client Compiler for Java 6. *ACM Trans. Archit. Code Optim.* 5, 1, Article 7 (May 2008), 32 pages. https://doi.org/10.1145/1369396.1370017

[39] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

[40] 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 2018 International Symposium on Code Generation and Optimization (CGO 2018)*. ACM, New York, NY, USA, 126–137. https://doi.org/10.1145/3168811

[41] Sheng Liang and Gilad Bracha. 1998. Dynamic Class Loading in the Java Virtual Machine. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '98)*. ACM, New York, NY, USA, 36–44. https://doi.org/10.1145/286936.286945

[42] Paul Lokuciejewski and Peter Marwedel. 2011. *Worst-Case Execution Time Aware Compilation Techniques for Real-Time Systems*. Springer Netherlands. https://doi.org/10.1007/978-90-481-9929-7_4

[43] Conor Mcbride and Ross Paterson. 2008. Applicative Programming with Effects. *J. Funct. Program.* 18, 1 (Jan. 2008), 1–13. https://doi.org/10.1017/S0956796807006326

[44] Scott McFarling. 1991. Procedure Merging with In-

struction Caches. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI '91)*. ACM, New York, NY, USA, 71–79. https://doi.org/10.1145/113445.113452

[45] Erik Meijer. 2012. Your Mouse is a Database. *Commun. ACM* 55, 5 (May 2012), 66–73. https://doi.org/10.1145/2160718.2160735

[46] Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java hotspotTM Server Compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1 (JVM'01)*. USENIX Association, Berkeley, CA, USA, 1–1. http://dl.acm.org/citation.cfm?id=1267847.1267848

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

[48] Aleksandar Prokopec. 2014. *Data Structures and Algorithms for Data-Parallel Computing in a Managed Runtime*. Ph.D. Dissertation. IC, Lausanne. https://doi.org/10.5075/epfl-thesis-6264

[49] Aleksandar Prokopec. 2015. SnapQueue: Lock-free Queue with Constant Time Snapshots. In *Proceedings of the 6th ACM SIGPLAN Symposium on Scala (SCALA 2015)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2774975.2774976

[50] Aleksandar Prokopec. 2016. Pluggable Scheduling for the Reactor Programming Model. In *AGERE!* 41–50.

[51] Aleksandar Prokopec. 2017. Accelerating by Idling: How Speculative Delays Improve Performance of Message-Oriented Systems. In *Euro-Par 2017: Parallel Processing*, Francisco F. Rivera, Tomás F. Pena, and José C. Cabaleiro (Eds.). Springer International Publishing, Cham, 177–191.

[52] Aleksandar Prokopec. 2017. Analysis of Concurrent Lock-Free Hash Tries with Constant-Time Operations. *ArXiv e-prints* (Dec. 2017). arXiv:cs.DS/1712.09636

[53] Aleksandar Prokopec. 2017. Encoding the Building Blocks of Communication. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2017)*. ACM, New York, NY, USA, 104–118. https://doi.org/10.1145/3133850.3133865

[54] Aleksandar Prokopec. 2018. Artifact Evaluation for the Optimization-Driven Incremental Inline Substitution Algorithm for Just-In-Time Compilers. (Dec. 2018). https://doi.org/10.5281/zenodo.2328430

[55] Aleksandar Prokopec. 2018. Cache-Tries: Concurrent Lock-Free Hash Tries with Constant-Time Operations. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18)*. ACM, New York, NY, USA, 15. https://doi.org/10.1145/3178487.3178498

[56] Aleksandar Prokopec. 2018. Efficient Lock-Free Removing and Compaction for the Cache-Trie Data Structure. In *Euro-Par 2018: Parallel Processing - 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings*. 575–589. https://doi.org/10.1007/978-3-319-96983-1_41

[57] Aleksandar Prokopec. 2018. Efficient Lock-Free Removing and Compaction for the Cache-Trie Data Structure. https://doi.org/10.6084/m9.figshare.6369134. (2018).

[58] Aleksandar Prokopec. 2018. Reactors.IO Website. (2018). http://reactors.io

[59] Aleksandar Prokopec, Phil Bagwell, and Martin Odersky. 2011. *Cache-Aware Lock-Free Concurrent Hash Tries*. Technical Report.

[60] Aleksandar Prokopec, Phil Bagwell, and Martin Odersky. 2011. *Lock-Free Resizeable Concurrent Tries*. Springer Berlin Heidelberg, Berlin, Heidelberg, 156–170. https://doi.org/10.1007/978-3-642-36036-7_11

[61] Aleksandar Prokopec, Phil Bagwell, Tiark Rompf, and Martin Odersky. 2011. A Generic Parallel Collection Framework. In *Proceedings of the 17th international conference on Parallel processing - Volume Part II (Euro-Par'11)*. Springer-Verlag, Berlin, Heidelberg, 136–147. http://dl.acm.org/citation.cfm?id=2033408.2033425

[62] Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. 2012. Concurrent Tries with Efficient Non-blocking Snapshots. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. ACM, New York, NY, USA, 151–160. https://doi.org/10.1145/2145816.2145836

[63] Aleksandar Prokopec, Philipp Haller, and Martin Odersky. 2014. Containers and Aggregates, Mutators and Isolates for Reactive Programming. In *Proceedings of the Fifth Annual Scala Workshop (SCALA '14)*. ACM, New York, NY, USA, 51–61. https://doi.org/10.1145/2637647.2637656

[64] 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 2017)*. ACM, New York, NY, USA, 29–40. https://doi.org/10.1145/3136000.3136002

[65] Aleksandar Prokopec and Fengyun Liu. 2018. On the Soundness of Coroutines with Snapshots. *CoRR* abs/1806.01405 (2018). arXiv:1806.01405 http://arxiv.org/abs/1806.01405

[66] Aleksandar Prokopec and Fengyun Liu. 2018. Theory and Practice of Coroutines with Snapshots. In *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*. 3:1–3:32. https://doi.org/10.4230/LIPIcs.ECOOP.2018.3

[67] Aleksandar Prokopec, Heather Miller, Philipp Haller, Tobias Schlatter, and Martin Odersky. 2012. *FlowPools: A Lock-Free Deterministic Concurrent Dataflow Abstraction, Proofs*. Technical Report.

[68] Aleksandar Prokopec, Heather Miller, Tobias Schlatter, Philipp Haller, and Martin Odersky. 2012. FlowPools: A Lock-Free Deterministic Concurrent Dataflow Abstraction. In *LCPC*. 158–173.

[69] Aleksandar Prokopec and Martin Odersky. 2014. Near Optimal Work-Stealing Tree Scheduler for Highly Irregular Data-Parallel Workloads. In *Languages and Compilers for Parallel Computing*, Călin Cascaval and Pablo Montesinos (Eds.). Springer International Publishing, Cham, 55–86.

[70] Aleksandar Prokopec and Martin Odersky. 2015. Isolates, Channels, and Event Streams for Composable Distributed Programming. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!) (Onward! 2015)*. ACM, New York, NY, USA, 171–182. https://doi.org/10.1145/2814228.2814245

[71] Aleksandar Prokopec and Martin Odersky. 2016. *Conc-Trees for Functional and Parallel Programming*. Springer International Publishing, Cham, 254–268. https://doi.org/10.1007/978-3-319-29778-1_16

[72] Aleksandar Prokopec, Dmitry Petrashko, and Martin Odersky. 2014. On Lock-Free Work-stealing Iterators for Parallel Data Structures. (2014), 10.

[73] B. G. Ryder. 1979. Constructing the Call Graph of a Program. *IEEE Trans. Softw. Eng.* 5, 3 (May 1979), 216–226. https://doi.org/10.1109/TSE.1979.234183

[74] Robert W. Scheifler. 1977. An Analysis of Inline Substitution for a Structured Programming Language. *Commun. ACM* 20, 9 (Sept. 1977), 647–654. https://doi.org/10.1145/359810.359830

[75] Tobias Schlatter, Aleksandar Prokopec, Heather Miller, Philipp Haller, and Martin Odersky. 2012. Multi-Lane FlowPools: A Detailed Look. (2012), 13.

[76] Manuel Serrano. 1997. *Inline expansion: When and how?* Springer Berlin Heidelberg, Berlin, Heidelberg, 143–157. https://doi.org/10.1007/BFb0033842

[77] 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, USA, 317–328. https://doi.org/10.1145/2095050.2095102

[78] 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. *SIGPLAN Not.* 46, 10 (Oct. 2011), 657–676. https://doi.org/10.1145/2076021.2048118

[79] Douglas Simon, John Cavazos, Christian Wimmer, and Sameer Kulkarni. 2013. Automatic Construction of Inlining Heuristics Using Machine Learning. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (CGO '13)*. IEEE Computer Society, Washington, DC, USA, 1–12. https://doi.org/10.1109/CGO.2013.6495004

[80] Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck, Thomas Würthinger, and Doug Simon. 2013. An Experimental Study of the Influence of Dynamic Compiler Optimizations on Scala Performance. In *Proceedings of the 4th Workshop on Scala (SCALA '13)*. ACM, New York, NY, USA, Article 9, 8 pages. https://doi.org/10.1145/2489837.2489846

[81] 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, USA, Article 165, 10 pages. https://doi.org/10.1145/2544137.2544157

[82] 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, USA, 221–226. https://doi.org/10.1145/1294325.1294356

[83] Arvind K. Sujeeth, Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, Victoria Popic, Michael Wu, Aleksandar Prokopec, Vojin Jovanovic, Martin Odersky, and Kunle Olukotun. 2013. Composition and Reuse with Compiled Domain-Specific Languages. In *ECOOP 2013 – Object-Oriented Programming*, Giuseppe Castagna (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 52–78.

[84] Spyridon Triantafyllis, Matthew J. Bridges, Easwaran Raman, Guilherme Ottoni, and David I. August. 2006. A Framework for Unrestricted Whole-program Optimization. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 61–71. https://doi.org/10.1145/1133981.1133989

[85] Oscar Waddell and R. Kent Dybvig. 1997. *Fast and effective procedure inlining*. Springer Berlin Heidelberg, Berlin, Heidelberg, 35–52. https://doi.org/10.1007/BFb0032732

[86] Philip Wadler. 1995. Monads for Functional Programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. Springer-Verlag, Berlin, Heidelberg, 24–52. http://dl.acm.org/citation.cfm?id=647698.734146

[87] Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant Propagation with Conditional Branches. *ACM Trans. Program. Lang. Syst.* 13, 2 (April 1991), 181–210. https://doi.org/10.1145/103135.103136

[88] Christian Wimmer and Hanspeter Mössenböck. 2005. Optimized Interval Splitting in a Linear Scan Register Allocator. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE '05)*. ACM, New York, NY, USA, 132–141. https://doi.org/10.1145/1064979.1064998

[89] 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 2017). ACM, New York, NY, USA, 662–676. https://doi.org/10.1145/3062341.3062381

[90] Graham Yiu. 2017. Partial Inlining with multi-region outlining based on PGO information. (2017). https://reviews.llvm.org/D38190 LLVM Pull Request.

[91] Graham Yiu. 2017. [RFC] Enhance Partial Inliner by using a general outlining scheme for cold blocks. (2017). https://reviews.llvm.org/D38190 LLVM Mailing List.

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