Inlining-Benefit Prediction with Interprocedural Partial Escape Analysis

Matthew Edwin Weingarten*
Oracle Labs
Zurich, Switzerland
matthew.weingarten@inf.ethz.ch

Theodoros Theodoridis
Swiss Federal Institute of Technology
Zurich, Switzerland
theodoros.theodoridis@inf.ethz.ch

Aleksandar Prokopec
Oracle Labs
Zurich, Switzerland
aleksandar.prokopec@oracle.com

Abstract

Inlining is the primary facilitating mechanism for *intraprocedural* Partial Escape Analysis (PEA), which allows for the removal of object allocations on a branch-by-branch basis and is critical for performance in object-oriented languages. Prior work used *interprocedural* Escape Analysis to make inlining decisions, but it discarded control-flow-sensitivity when crossing procedure boundaries, and did not weigh other metrics to model the cost-benefit of inlining, resulting in unpredictable inlining decisions and suboptimal performance. Our work addresses these issues and introduces a novel Interprocedural Partial Escape Analysis algorithm (IPEA) to predict the inlining benefits, and improve the cost-benefit model of an existing optimization-driven inliner. We evaluate the implementation of IPEA in GraalVM Native Image, on industry-standard benchmark suites Dacapo, ScalaBench, and Renaissance. Out of 36 benchmarks with a geometric mean runtime improvement of 1.79%, 6 benchmarks achieve an improvement of over 5% with a geomean of 9.10% and up to 24.62%, while also reducing code size and compilation times compared to existing approaches.

CCS Concepts: • **Software and its engineering → Source code generation**; Runtime environments; Just-in-time compilers.

Keywords: escape analysis, inlining, compilers, graalvm

ACM Reference Format:
Matthew Edwin Weingarten, Theodoros Theodoridis, and Aleksandar Prokopec. 2022. Inlining-Benefit Prediction with Interprocedural Partial Escape Analysis. In *Proceedings of the 14th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '22), December 05, 2022, Auckland, New Zealand*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3563838.3567677

*Also with Swiss Federal Institute of Technology.

VMIL '22, December 05, 2022, Auckland, New Zealand

ACM ISBN 978-1-4503-9912-8/22/12.
https://doi.org/10.1145/3563838.3567677

1 Introduction

Partial Escape Analysis (PEA) [50] is a control flow sensitive variant of Escape Analysis [7–9, 11, 12, 18, 25, 44, 55]. The analysis computes whether an object escapes outside the scope of the procedure and performs optimizations such as stack allocation for objects, replacing object fields with scalars, and eliding locks. Flow-sensitivity significantly increases the escape-analysis effectiveness, and decreases the dynamic allocation rate compared to flow-insensitive analysis [50]. This is achieved by allocating, or *materializing*, objects on some control-flow paths and keeping them *virtual* on others. PEA is an intraprocedural phase, so its potency is limited by calls to other procedures since almost all objects escape into at least one other procedure [25, 50].

In existing work, this limitation is mitigated with interprocedural analysis that foregoes control-flow sensitivity when crossing procedure boundaries [7, 11, 25]. The results of the analysis are then used to either guide inlining decisions [25] and then applying PEA within the scope of inlined procedures, or to employ alternative schemes such as stack allocation or region-based memory management, but without doing the inlining [11].

Although not mutually exclusive with interprocedural analyses, intraprocedural analysis is the more pragmatic choice for most compiler optimizations. In most optimizing compilers, optimization passes are intraprocedural [1, 15, 27, 32], and inlining kickstarts their symbiotic interplay. This is particularly prevalent for the optimizations performed by PEA, as one can easily see how scalar replacement consequently enables constant folding, conditional branch elimination, value numbering, and many others [36]. Most compilers therefore perform inlining as one of the earliest optimization passes, before optimizations such as PEA.

While delegating the responsibility of interprocedural escape analysis to inlining is powerful, it introduces its own set of challenges. Inlining can be modeled as the NP-hard Knapsack problem if a *cost-benefit* assignment is available at each callsite [45]. However, the cost-benefit assignment is a best-effort approximation of a ground-truth cost-benefit – to correctly assess cost-benefit, the compiler must predict the optimizations that follow inlining, either by inferring them from the procedure body [4, 46], or by using inlining trials [13]. Profiling data, such as the callsite frequencies and

call-target distributions, is then used to further assess the performance impact of each inlining decision [20, 21, 35].

Most prior work in the domain of inlining and interprocedural escape analysis indiscriminately applies ad-hoc heuristics to make inlining decisions whenever an allocation can be removed [5, 8, 25, 26, 33], and without ostensibly weighing these decisions against other considerations such as code size, compilation unit clustering, and instruction cost, which is what inliners traditionally do [2–4, 13, 35, 45, 46, 51]. This leads to inlining decisions that are based on a fragmented view of available information, leading to inconsistent performance metrics at best and adverse effects at worst.

This paper aims to address the following question – *how can interprocedural partial escape analysis improve inlining decisions, and consequently peak performance of user programs?* The contributions in the paper are as follows:

- A novel approach that interleaves interprocedural partial escape analysis with inlining and computes the allocation frequency of escaping objects, with the aim of improving inlining decisions. The approach maintains flow-sensitivity across procedure boundaries and utilizes the collected profiling information (Section 3.1).
- Integration of materialization frequency metrics (Section 3.2) into an existing optimization-driven incremental inlining algorithm [35] used in GraalVM Native Image, yielding a single point of accountability for inlining (Section 3.3).
- A production-ready implementation in state-of-the-art optimizing compiler Graal [15, 16], in ahead-of-time compilation mode on GraalVM Native Image [57].
- A comprehensive performance evaluation that consists of peak-performance, allocation-rate, code-size and compile-time analysis on DaCapo [6], Scalabench [47], and Renaissance benchmark suites [39–41], as well as the extensive tuning of the proposed heuristics. We report a geometric mean performance improvement of 1.79% up to 24.62% across all 36 benchmarks, and geomean code size improvement of 1.81%. In total there are 6 benchmarks with >5% speedup and geometric mean of 9.10% (Section 4).

2 Motivation & Background

Consider the method `put` of the `IntListMap` class in Listing 1, which closely corresponds to the class `ListHashMap` from the Groovy core library [19]. This collection stores key-value pairs as two consecutive entries in an array-based `list`. Keys are always primitive integers, but values are of any generic object type `V`. The `list` array is of `Object[]` type because JVM cannot store primitives and objects in the same array. Thus, a key must be *boxed* in line 6 into an object, before being stored. In the loop in line 7, `put` then scans the array up to position `last`, to see if the specified key is already in `list`. If an equal key is present at `i`, the existing value at `i + 1` is overwritten, but the key itself is not stored. If the key is not present, the `boxed` key is stored in line 10.

```
1  class IntListMap<V> {
2   Object[] list;
3   int last;
4
5   V put(int key, V val) {
6    var boxed = new Integer(key);
7    for (int i=0;i<last;i+=2)
8     if (list[i].equals(boxed))
9      return list[i + 1] = val;
10    list[last++] = boxed;
11    return list[last++] = val;
12   }
13 }
```

Listing 1. `IntListMap`

It seems as if the allocation can be delayed until line 10 – if a map kept reusing the same keys, this would reduce the allocation rate greatly. But alas – `boxed` escapes earlier, in the `equals` call in line 8! Luckily, any decent inliner will exploit the receiver-type profile to decide that `equals` always calls `Integer#equals` [14] – it inserts a speculative typecheck, and trivially inlines `Integer#equals`, which compares the primitive `integer` fields of the two boxed `Integer` objects:

```
if (!(list[i] instanceof Integer)) deopt();
if (list[i].integer==boxed.integer) return list[i + 1] = val;
```

If the dynamic typecheck fails, the code is either deoptimized, or the execution gets continued in *baseline* compiled code, which does not inline `equals`, but is less efficient [56].

```
1 for (i = 0; i < last; i += 2)
2  if (list[i].integer == key)
3   return list[i + 1] = val;
4 var boxed1 = new Integer(key);
5 list[last++] = boxed1;
6 return list[last++] = val;
```

Listing 2. PEA on `put`

PEA can now postpone the `boxed` allocation – it keeps it virtual in line 2 in Listing 2, where the read `boxed.integer` can be replaced with just `key`; and materializes it in line 4. Yet, a more realistic list-map, one that better resembles Groovy's `ListHashMap` [19], would keep its key type generic.

```
class ListMap<K, V>
```

In this case, the `key` parameter has a type `K`, which is erased to an `Object` type [10] – a primitive integer key must be boxed at `put`'s callsite, before `put` is even called, so there is nothing PEA can do! The only hope is that the inliner decides to inline `put` wherever `put` is called – then PEA could 'see' the allocation of `key`. The following procedure inserts the primitive key `1` into a `ListMap` that maps `Integer` keys to `String` values:

```
1 V put(K key, V val) {
2  for (int i=0;i<last;i+=2)
3   if (list[i].equals(key))
4    return list[i + 1] = val;
5   list[last++] = key;
6   return list[last++] = val;
7 }
```

Listing 3. Generic `put`

```
void client(ListMap<Integer, String> m)
  { return m.put(1, "one"); }
```

Since the key-type is `Integer`, the `javac` compiler 'autoboxes' the primitive `int` value that is passed in the `put` call. The actual compiled code resembles the following:

```
return m.put(new Integer(1), "one");
```

If something were to 'convince' the inliner that there is benefit in inlining the `put` call, the resulting compilation unit would contain both the allocation and its usages. This extended horizon over the program would allow PEA to transform the code to the following:

```
for (i = 0; i < m.last; i += 2)
  if (m.list[i].integer == 1) return m.list[i + 1] = "one";
var boxed1 = new Integer(1);
m.list[m.last++] = boxed1;
return m.list[m.last++] = "one";
```

But, despite having delayed the allocation of the `Integer`, was inlining `put` really worth it? It depends. For one thing, the `put` call inside the `client` procedure should be overall executed often, otherwise, the gains are anyway miniscule. Then, we should be reasonably assured that path to the final `return` is cold, that is, that most of the time we exit early, in the `for` loop. Finally, we should make sure that there are no other 'better' inlining candidates – perhaps the inlining budget would be better spent elsewhere?

To deal with such considerations, inliners build cost-benefit models that guide their decisions. These models are usually based on execution profiles, knowledge about code-size and opportunities for other optimizations. Rather than disregarding all these other factors, perhaps we should try to influence an existing inliner by biasing its cost-benefit model?

This deliberation motivates our main research questions:

- **RQ1:** How do we compute the frequency of materializations for objects that escape across procedures?
- **RQ2:** How can we quantify the benefit of reducing materialization-frequency through inlining, such that we improve an existing cost-benefit estimation?

2.1 Intraprocedural Partial Escape Analysis

Let's consider how standard intraprocedural PEA works when viewed at the source code level of the previous example. At each instruction n, PEA maintains a mapping $O(n)$ called *virtualization*, which maps each value to the virtualization state of that value. Each allocated object is mapped to either a materialized value $m \in \mathbb{V}$, indicating that the respective object escaped and was allocated as another value m; or to a sequence of field-value pairs $f_j \rightarrow x_j$, indicating the state of an object that did not escape (i.e. object that is virtual):

$$O(n) : \mathbb{V} \mapsto \mathbb{V} \cup \{(f_0 \rightarrow x_0, \ldots, f_i \rightarrow x_i) : f_j \in \mathbb{S} \wedge x_j \in \mathbb{V}\}$$

Above, $\mathbb{V}$ represents the set of referable values in the program, f_j represents a field name from the set of all strings $\mathbb{S}$, and each field is mapped to another value $x_j \in \mathbb{V}$.

In our example with the `put(int, V)` procedure, after executing the statement $n_0 \equiv$`var boxed=new Integer(key)` in line 6, O maps the `boxed` value to a tuple whose `integer` field is set to the value of the parameter `key`:

$$O(n_0) = \{\texttt{boxed} \mapsto (\texttt{integer} \rightarrow \texttt{key})\}$$

Statements in line 7 and line 8 do not modify the `boxed` object, in other words $O(n_1) = O(n_0) = O(n_2)$. However, n_2 can exploit the mapping O, namely, n_2 can be replaced with `list[i].integer==key`, because $O(n_2)$ tells us that `boxed` is virtual (i.e. did not yet escape), and its `value` field is equal to `key`. The next statement $n_3 \equiv$`list[last++]=boxed` materializes the `boxed` object, since the reference to `boxed` escapes as part of the array-write. Mapping $O(n_3)$ becomes:

$$O(n_3) = \{\texttt{boxed} \mapsto \texttt{boxed1}\}$$

which means that original allocation `boxed` is on this control-flow path materialized under the name `boxed1`. PEA thus inserts the instruction `var boxed1=new Integer(key)` just before n_3, and makes sure that the `boxed1` value is used instead of `boxed` on the rest of the control-flow path.

PEA on control-flow graphs. Having grasped how PEA uses the mapping O, let's express PEA as a dataflow analysis on control-flow graphs (CFGs). Control-flow graphs will allow us to more concisely define how individual instructions affect the mapping O. Also, our actual implementation was done on Graal IR, which is conceptually a reducible control-flow graph. The CFG that corresponds to the `put` procedure from Listing 1 is shown in Figure 1.

In the previous discussion of Listing 1, we were somewhat lax about what an *instruction* is – in the CFG, we define an individual instruction as a node. Nodes may be connected with directed edges, which represent the control-flow of the program. For each type of node, we can define two mappings O_{in} and O_{out}, as the virtualization before and after the respective node is executed, and use them to describe how each node changes the virtualization.

Let the set $\mathbb{A}$ be the set of all nodes that represent object allocations. Then, node $a \in \mathbb{A}$, which allocates an object with i fields, modifies O as follows ($\perp$ is the default value):

$$O_{out}(a) = O_{in}(a)[a \mapsto (f_0 \rightarrow \perp, \ldots, f_i \rightarrow \perp)] \quad a \in \mathbb{A}$$

Next, let $\mathbb{W}$ be the set of nodes that store a value x into a field f of an object allocated at a. For a node $w^a_{f,x} \in \mathbb{W}$ we have two cases – the field f is updated to x only if a is virtual, and otherwise, the mapping is not modified:

$$O_{out}(w^a_{f,x}) = \begin{cases} O_{in}(w^a_{f,x})[a \mapsto (\ldots, f \rightarrow x, \ldots)] \\ \quad \textit{if } a \mapsto (\ldots, f \rightarrow y, \ldots) \in O_{in}(w^a_{f,x}) \\ O_{in}(w^a_{f,x}) \quad \textit{otherwise when } w^a_{f,x} \in \mathbb{W} \end{cases}$$

Set $\mathbb{E}$ consists of nodes that cause an object to escape to the heap: in intraprocedural PEA, these are calls to other subroutines, writes to global fields, and writes to already materialized objects. We use the relation $n \xrightarrow{\text{esc}} a$ to denote that a node n causes an object allocated at node a to escape.

Let $e^a \xrightarrow{\text{esc}} a$. Again, O changes only if object a is virtual, in which case a is materialized. For all fields f, if f points to another virtual object, that object is transitively materialized:

$$\mu(O, x) = \begin{cases} \mu(\ldots \mu(O[x \mapsto m], x_0), \ldots, x_i) \\ \quad \textit{if } x \mapsto (f_0 \rightarrow x_0, \ldots, f_i \rightarrow x_i) \in O \\ O \quad \textit{otherwise} \end{cases}$$

$$O_{out}(e^a) = \mu(O_{in}(e^a), a) \quad \textit{when } e^a \xrightarrow{\text{esc}} a$$

Above, the helper function μ recursively traverses the object graph. Each virtual object is replaced with m, which represents a newly inserted node that *materializes* the object that had been allocated at node a in the original program.

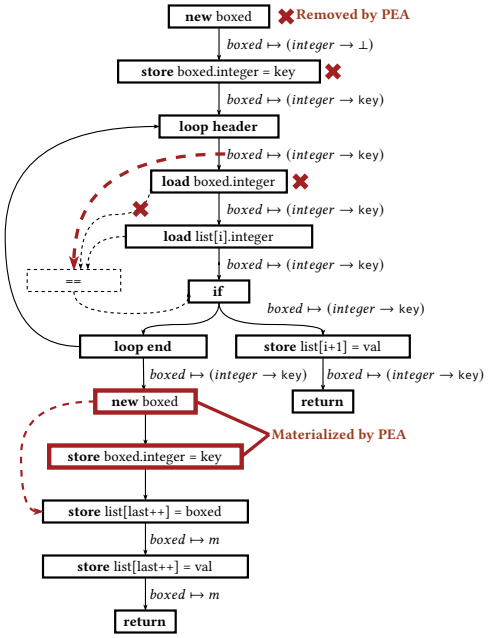


Figure 1. Simplified control flow(⟶) and data flow (⇢) of `put`. The boxed allocation is moved to latest point before `boxed` becomes materialized and loads are replaced with scalars from the virtual state.

One may now ask: could the escaping node e^a be used to represent the materialization too, i.e. is $m = e^a$? This is not generally the case – to see why, consider the *merge* node, which has multiple predecessors. A merge node j can implicitly trigger an escape in order to enforce a consistent state in its successor node. First, a value only survives the merge if it exists in all predecessor states. Then, if an object is virtual in all predecessors, it can remain virtual after the merge. But, if the object is materialized in at least one predecessor, then it must be materialized in the others too (for simplicity, this is what most PEA implementations do). Formally:

$$\iota(x,y) = \begin{cases} (f_1 \to \iota(x_1,y_1), \dots, f_n \to \iota(x_n,y_n)) \\ \quad if\, x = (f_1 \to x_1, \dots, f_n \to x_n) \\ \quad \wedge y = (f_1 \to y_1, \dots, f_n \to y_n) \\ \varphi(m_1,m_2) \quad otherwise \end{cases}$$

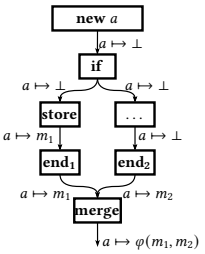
$$O_1 \sqcap O_2 = [a \mapsto \iota(x,y) | \forall a.a \mapsto x \in O_1 \wedge a \mapsto y \in O_2]$$

$$O_{out}(j) = O_{in,1}(j) \sqcap \dots \sqcap O_{in,p}(j)$$

Above, for brevity, φ is a *Phi function* from static single-assignment (SSA) form, which takes a different value depending on the predecessor by which the merge was reached – it can be trivially translated to a CFG with variables [43].

In the definition of ι, if the allocation was materialized as m_1 on one predecessor, and virtual on the second predecessor, then it must be materialized to a node m_2 on the second predecessor too. Observe the CFG with a merge node. The allocation a escapes in the left branch at the `store` node, but not on the right branch. Here, *only* `store` $\xrightarrow{\text{esc}}$ a, but *both* `store` $\xrightarrow{\text{mat}}$ a *and* `end`$_2$ $\xrightarrow{\text{mat}}$ a, so allocations are inserted in both branches.

We can now inspect how the virtualization O changes at the edges between nodes in the CFG of the `put` procedure in Figure 1, The object behind the `boxed` allocation is virtual throughout the loop, and allows simplifying the check inside the loop. The store node causes the object to escape after the loop exit ($\xrightarrow{\text{esc}}$), and the newly inserted red/bold allocation node is that allocation's materialization node ($\xrightarrow{\text{mat}}$).



3 Algorithm Description

The main idea behind our Interprocedural Partial Escape Analysis (IPEA) algorithm is to predict the materialization frequency after inlining, and use it to adjust the inlining-benefit approximation of a callee. The frequency of materializations determines the number of dynamic allocations at runtime. We first discuss how IPEA locates materializations across the entire call tree instead of inside a single procedure.

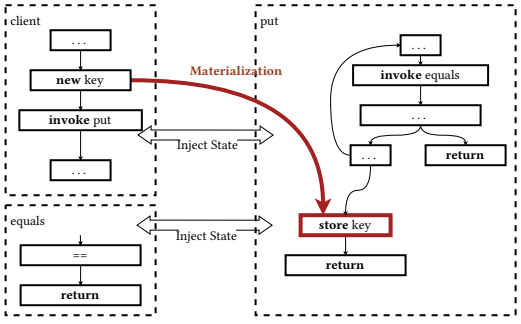


Figure 2. Call tree representation of `put` and client code and `key` pointing to its interprocedural materialization.

3.1 Interprocedural Partial Escape Analysis

Interprocedural analysis accounts for the relationships between procedures, so we shift our perspective from an individual procedure body to a set of procedure bodies logically assembled in *call tree* representation. Revisiting our generic `ListMap#put` procedure from Section 2, Figure 2 now shows the call tree of `client`, `put` and equals and their respective simplified Intermediate Representations (IR). Recall that PEA could postpone the materialization of the `key` parameter of `put` *after* the loop and before it is written to the array. In order to find this materialization location, the virtualization states must be shared across procedure boundaries. We call this *virtual state injection*, or *virtual injection*.

In the example in Figure 2, the parameter `key` of the `put` procedure originates from the allocation in the client code. To simulate the effects of inlining, we can intuitively treat the parameter node as an extension of the allocation in the caller.

If `key` is still virtual at the callsite, we can replace the parameter with a virtual object originating from the allocation in the client. From this point on, when a materialization is triggered for the replaced parameter node, we know that this is exactly where PEA will insert the allocation after inlining.

Similarly, when the entire procedure body is completely processed, the callsite must reacquire the virtual state from its callee, i.e. the virtual state must be re-injected back into the caller. Let's assume the `equals` procedure has yet to be inlined – the `key` object is also passed to the `equals` procedure inside `put`. If the `equals` procedure materializes its parameter, there is no benefit in inlining `put`. Our analysis must therefore first process the entire `equals` procedure, and re-inject the virtual state into `put` to determine that `key` would stay virtual until `key` is materialized by the store node.

Having provided an intuition of IPEA on the client code, we now detail how the intraprocedural analysis is extended to allow the prediction of materializations after inlining. To properly model O across procedure boundaries, invoke nodes no longer cause escaping – instead, invokes trigger a recursive analysis in the subtree of the call tree. All other nodes that trigger escaping are treated as in ordinary PEA.

We also add an additional state component on top of the local analysis state O_n, namely the shared global state *materializations* $\mathcal{M}$, which maps each allocation in the call tree to the set of its materializations in the call tree. Our new compositional state $\mathcal{S}$ is now defined as follows:

$$\mathcal{S}(n) = \langle O(n), \mathcal{M} \rangle$$

$$\mathcal{M} : \mathbb{A} \rightarrow \{n \mid n \xrightarrow{\text{mat}} a \wedge a \in \mathbb{A}\}$$

where $O(n)$ is a per-node state, and $\mathcal{M}$ is a shared global state across the entire call tree. $\mathcal{M}$ holds the desired analysis results, and is later used to drive inlining decisions. When materializations are determined, i.e. after IPEA finishes processing a procedure (a call-tree node), for any node n such that $n \xrightarrow{\text{mat}} a$, we update $\mathcal{M}$ accordingly:

$$\mathcal{M}[a] = \mathcal{M}[a] \cup \{n\}$$

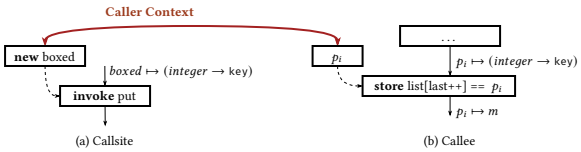
To formalize the notion of the call tree, we rely on the relation $c_{a_0,\ldots,a_j} \xrightarrow{\text{calls}} p_0, \ldots, p_j$, which denotes that an invoke node $c_{a_0,\ldots,a_j}$ with arguments $a_0, \ldots, a_j$ calls the callee procedure whose parameters are nodes p_0 to p_j.

State-update rules. IPEA processes specific nodes, namely the `invoke`, `parameter` and `return` nodes, in a different way. When we encounter an invoke node $c_{a_0,\ldots,a_j}$ with arguments $a_0, \ldots, a_j$ such that $c_{a_0,\ldots,a_j} \xrightarrow{\text{calls}} p_0, \ldots, p_j$, we define the update rules in the callee as follows:

$$O_{out}(p_i) = \begin{cases} O_{in}(p_i)[p_i \mapsto (f_0 \rightarrow x_0, \ldots, f_k \rightarrow x_k)] \\ \quad \text{if } a_i \mapsto (f_0 \rightarrow x_0, \ldots, f_k \rightarrow x_k) \in O_{in}(c_{a_0,\ldots,a_j}) \\ O_{in}(p_i) \quad \text{otherwise when } c_{a_0,\ldots,a_j} \xrightarrow{\text{calls}} p_0, \ldots, p_j \end{cases}$$

The above states that the initial state of a formal parameter is virtualized if the corresponding argument in the caller remained virtual, and materialized otherwise.

The following illustrates the rule on the `put` procedure:



At the callsite, the first argument is mapped to the `new boxed` node, so the corresponding formal parameter in the callee is mapped accordingly. After the array-write is processed, the materialization map will contain a single mapping $\mathcal{M} \equiv \{\texttt{boxed} \mapsto \{\texttt{store list}\}\}$, as indicated in Figure 2.

Let the set $\mathbb{R}$ of *return nodes* consist of nodes r_x, such that the return node r_x returns the value of node x. Handling a return node r_x serves two purposes. The first is to re-inject the state of the callsite arguments back into the caller. If a formal parameter gets materialized, the callsite argument must also be materialized. Second, if a virtual object is returned from the callee, the invoke node can be mapped to a virtual object in the caller thereafter.

We use a helper function π to capture how the virtualization state of the parameter p_i affects the callee state O:

$$\pi(O, p_i) = O[a_i \mapsto v] \;\; \textit{such that} \;\; \exists c_{a_0,\ldots,a_j} \xrightarrow{\text{calls}} p_0, \ldots, p_j \wedge p_i \mapsto v \in O_{out}(r_x) \wedge r_x \in \mathbb{R}$$

The above says that for any parameter, there must be a return node that mentions that parameter's state, and that the corresponding callsite argument must be mapped accordingly.

We use another helper ϱ to capture how the return value alters the virtualization state of the callee:

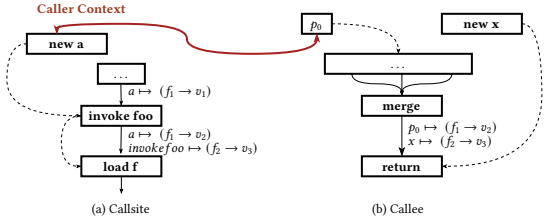
$$\varrho(O, c_{a_0,\ldots,a_j}) = O[c_{a_0,\ldots,a_j} \mapsto v] \quad c_{a_0,\ldots,a_j} \xrightarrow{\text{calls}} p_0, \ldots, p_j \wedge p_0, \ldots, p_j \in O_{out}(r_x) \wedge x \mapsto v \in O_{out}(r_x) \wedge r_x \in \mathbb{R}$$

The above says that for any invoke $c_{a_0,\ldots,a_j}$, there must be a return node r_x in the corresponding callee whose returned value x is mapped to some state v – the virtualization state of the invoke $c_{a_0,\ldots,a_j}$ must be remapped to that state v.

The rule for invokes is now concisely expressed as:

$$O_{out}(c_{a_0,\ldots,a_j}) = \varrho(\pi(\ldots \pi(O_{in}(c_{a_0,\ldots,a_j}), p_0), \ldots, p_j), c_{a_0,\ldots,a_j})$$

The following figure illustrates the previous rule – the state of the object a changes in the callee `foo` (but is not materialized), so the analysis continues with the new virtual state of a. Moreover, the return object x is created inside the scope of the callee, and remains virtual in the callee.



In the preceding rule, we implicitly assumed that there is a single return node r_x in any callee. Indeed, the functions π and ϱ are not defined when there are multiple return nodes. To address this, we simply transform the CFG of each callee

by replacing all the return nodes r_{x_i} with a merge node j followed by a single return node $r_{\varphi(x_0,\ldots,x_k)}$ (which, in fact, reflects what happens during inlining). This way, the virtualized state persists only if all return nodes return a virtualized object, as per the rule for merge nodes from Section 2.1.

Pseudocode. The code for the IPEA algorithm, which produces the map of materializations $\mathcal{M}$ for a call tree $\mathcal{T}$, is shown in Algorithm 1. The call `runIPEA(root(`$\mathcal{T}$`), {})` starts the algorithm. It traverses the CFG in a reverse postorder, using the extended set of rules for updating the virtualization state. If it encounters an invoke, it first calls `runIPEA` on the child node of the call tree before updating the state in the caller. In each call tree node, CFG traversal must be repeated until reaching a fix point [50]. Every time a call tree node reaches a fix point, the materialization map $\mathcal{M}$ is updated.

Algorithm 1: Interprocedural PEA

```
input  : Call tree T
output : Materializations M : a → {n | n --mat--> a ∧ a ∈ 𝔸}
Procedure runIPEA(CFG, M) is
    S_in(start) ← ⟨{}, M⟩
    while not fixedPointReached() do
        for node in reversePostOrder(CFG) do
            if node is Invoke then
                runIPEA(CFG(node), M)
            S_out(n) ← process(node, S_in(node))
    for (n,a) such that n --mat--> a do
        M[a] ← M[a] ∪ n
```

3.2 Materialization-Frequency Reduction

In our running example from Section 2, the reduction of `key` allocations depends on the workload in the `client` code – if the `client` only occasionally inserts a previously unseen key, then the number of allocations is reduced considerably. To reflect this in the inlining benefit, we use the materialization map $\mathcal{M}$ and node-frequency profiles to compute the *materialization-frequency reduction*:

$$\mathcal{FR}(a) = \frac{F(a) - \sum_{m \xrightarrow{\text{mat}} a} F(m)}{F(a)} \qquad \in [0,1] \qquad (1)$$

Above, $F(n)$ is the execution frequency of a node relative to the entry to the root compilation unit in the call tree. Note that the frequencies of materialization nodes are summed – correctness is given by guarantees a PEA invariant: For any two nodes n_1 and n_2 that materialize the same allocation node a, there is no execution path that passes both through n_1 and n_2. In other words, for a single a, every execution path has at most a single node n, such that $n \xrightarrow{\text{mat}} a$.

3.3 Integrating IPEA into the Incremental Inliner

We extended an existing implementation of the incremental inliner [35] with our analysis. We now present a simplified description of the existing inliner, and show how it was enhanced. The incremental inliner is representative of typical inlining algorithms, and we believe that our analysis can be easily applied to similar inliners.

Each time a method is submitted for compilation, that method becomes the root compilation unit of its respective call tree. The inliner incrementally *expands* the call tree during an *expansion phase*. Expanding means picking a leaf node in the call tree, and attaching the callees of that node as children to the call tree. A call-tree node is initially unexplored, and called a *cutoff node*. After the expansion phase, the inliner assigns cost-benefit to each call-tree node. Based on the cost-benefit, some call-tree nodes are inlined during the *inlining phase*, and the inliner subsequently repeats these stages. This cycle continues until reaching a budget limit [35].

IPEA as described in Algorithm 1 is inserted into the inliner after expansion and cost-benefit assignments, but before inlining (see Algorithm 2). We note that each invoke that corresponds to a cutoff node is considered an analysis boundary, and is treated as a regular escape.

Algorithm 2: Integration of IPEA

```
input  : Root compilation unit root
Procedure incrementalInliner(root) is
    while continueInlining() do
        expansion()
        M ← runIPEA(root)
        computeBenefits(root)
        for a ∈ AllocationNode do
            fr ← FR(a)
            boostSubtree(fr)
        inlining()
```

Consider the compilation of the `client` procedure from Section 2. Its inlining schedule will typically be as follows:

(i) Build initial call tree for `client`, add `put` as a cutoff.
(ii) Expand `put`, add `equals` as a cutoff node, expand `equals`.
(iii) Run IPEA on the call tree (`client`, `put`, and `equals`).
(iv) Compute cost-benefit of `put` and `equals`.
(v) Boost both `put` and `equals` based on the reduction in materialization frequency of `key`.
(vi) Decide to inline both `put` and `equals`.

Benefit. Before inlining, each node in the call tree is assigned a benefit. This benefit is a best-effort estimate of the performance gains from inlining, and it guides the inlining decisions. Our analysis adjusts the benefit assignment by incorporating materialization-frequency reduction. Our new benefit function is the following:

$$\mathcal{B}(c_{a_0,\ldots,a_j}) = \mathcal{B}'(c_{a_0,\ldots,a_j}) \cdot \max_{i \in 0,\ldots,j} (\delta_{mb})^{\mathcal{FR}(a_i)}$$

where $\mathcal{B}'(c_{a_0,\ldots,a_j})$ represents the previous benefit function for a node at callsite $c_{a_0,\ldots,a_j}$, and δ_{mb} is the *materialization boost*, a tunable hyperparameter that quantifies how much the inliner should value the reduction in materialization frequency. A call tree node is boosted relative to the frequency reduction $\mathcal{FR}(a)$ if a escapes into the invoke $c_{a_0,\ldots,a_j}$. For example, in the earlier `client` method, the boost $\mathcal{FR}$(`key`)

is applied to both `equals` and `put`, since `key` escapes to both, and inlining both of them is required to reduce allocations.

Expansion phase. In Algorithm 2, we applied `runIPEA` to the point after the expansion phase. However, since each expansion phase expands multiple cutoff nodes of the call tree, we can also incrementally run IPEA on the subtree affected by the expansion. The advantage of doing so is that we can boost the expansion priority of cutoffs at which objects escape – each cutoff node has an associated expansion priority, which decides how soon that node will be expanded.

During IPEA, we track an additional state component $\mathcal{E}$ for allocations that escape into cutoff nodes:

$$\mathcal{E} : \{c_{a_0,\dots,a_j} \textit{ is a cutoff}\} \rightarrow \{a_i | c_{a_0,\dots,a_j} \xrightarrow{\text{esc}} a_i\}$$

The cutoff-node priorities $\mathcal{P}$ are then continuously updated according to the following equation:

$$\mathcal{P}(n) = \mathcal{P}'(n) \cdot \delta_{eb} \cdot |\mathcal{E}(n)|$$

where the new priority $\mathcal{P}$ boosts the vanilla (cost-benefit-based [35]) priority $\mathcal{P}'$ by the number of escaping objects. This equation also introduces an additional hyperparameter δ_{eb}, or *escape boost*, which determines the expansion priority given to virtual objects that are escaping.

Cutoff weight. In an idealized analysis, the entire call tree is available, and no invoke causes an object to escape. However, in practice, expansion budget is often insufficient, and the call tree is usually infinite due to recursion, meaning that at least some cutoff nodes remain in the call tree. Whenever an object escapes at a cutoff node, this can be for two reasons: either the object would really be materialized after expansion, or the object is materialized only because we did not expand the callee yet. To prevent over-penalizing such materializations, we found it useful to optimistically assume that an object that escapes at a cutoff may survive. We,therefore, updated the frequency-reduction expression from Equation (1) with the *cutoff weight* hyperparameter $\delta_{cw} \in [0, 1]$:

$$\mathcal{FR}(a) = \frac{F(a) - \sum_{m \xrightarrow{\text{mat}} a} \phi_m \cdot F(m)}{F(a)} \quad \phi_m = \begin{cases} \delta_{cw} & m \textit{ is cutoff} \\ 1 & \textit{otherwise} \end{cases}$$

We discuss the tuning of this heuristic in Section 4.

4 Evaluation

The goal of the evaluation is to show improvements over flow-insensitive escape analysis techniques in terms of runtime, code-size and compilation times, examine the profiled dynamic number of allocations, and the effect of heuristics.

We compare our Interprocedural PEA (IPEA) to a Base, which runs only the optimization-driven inliner [35], that is run in every configuration. Our second comparison is Partial Escape Selective Inlining (PESI), the control flow insensitive inlining technique implemented in GraalVM.

Decreasing the runtime is the principal goal of our analysis phase. We emphasize the GraalVM compiler has been already heavily optimized and most performance-critical methods are inlined. Nevertheless, we consider improvements of > 5% to be significant and valuable incremental contributions to GraalVM. While secondary to runtime performance, keeping code size low is meaningful for Native Image, especially for one of the main use cases of cloud deployment [49, 53]. Build time is an important measurement, as PEA is an expensive analysis for large procedures, even intraprocedurally.

Methodology. All reported results were run on an Intel(R) Xeon(R) E5-2699 v3 @2.30 GHz with 72 cores and 256GB of RAM. The benchmarks were bound to 36 CPUs on the same NUMA node and the maximum Heapsize was capped at 8GB. All benchmarks were run on JDK 17 except for the Renaissance-spark jobs.[1] Relative improvements between T_{old} and T_{new} are always computed as $\frac{T_{old}-T_{new}}{T_{old}}$ unless stated otherwise and averaged improvements use geometric mean.

Benchsuite	Version	JDK
Renaissance [40]	0.14.1	17
Renaissance-spark	0.11.0	11
Dacapo [6]	9.12 mr1	17
ScalaBench [47]	0.1.0	17

Runtimes. Plot in Figure 3 compares relative runtime performances across all benchmarks (lower is better). We include an additional column for IPEA OPT, where the best hyperparameters are chosen for each individual benchmark, instead of optimizing for the entire suite. We observe that some types of benchmarks react strongly to IPEA, where benchmarks improve over PESI, like `scalap` - 5.5%, `scalac` -5.74%, `finagle-chirper` -7.74% `avrora` -16.07% and `mnemonics` -24.62%. In practice, some benchmarks show no performance difference when optimizing for allocation reduction. This may be the case for more CPU-bound applications, or code sections that perform allocations are hot enough to be inlined based on frequency and code size metrics. Further, we see that individual benchmarks have more optimization opportunities: Benchmarks like `kiama`, `gauss-mix`, `par-mnemonics` and `fop` leave major improvements of up to 16% between IPEA and IPEA OPT unrealized. This suggests future work on this topic is of relevance. Some benchmarks exhibit regressions, `factorie` by +3.35%, `db-shootout` +5.07% and most notably `xalan` +16.07%. For both `db-shootout` and `factorie`, these regressions can be eliminated by choosing better parameters. We discuss `xalan` in more depth in the heuristic tuning section.

Code size. See Figure 4 for codesize and build time results. IPEA decreases code size almost across the board compared to PESI, with the only exceptions being `scalariform`, `scalaxb` and `scala-doku`. On average our code size has decreased by 1.81%. This suggests our analysis has increased inlining precision, were overall less, but more impactful procedures are inlined [2].

[1] Renaissance-spark benchmarks did not run on JDK 17 at time of experimentation:page-rank, movie-lens,db-shootout, neo4j-analytics, gauss-mix. rx-scrabble, dotty, naive-bayes, als.

[2] We stress that more inlining does not always cause larger code.

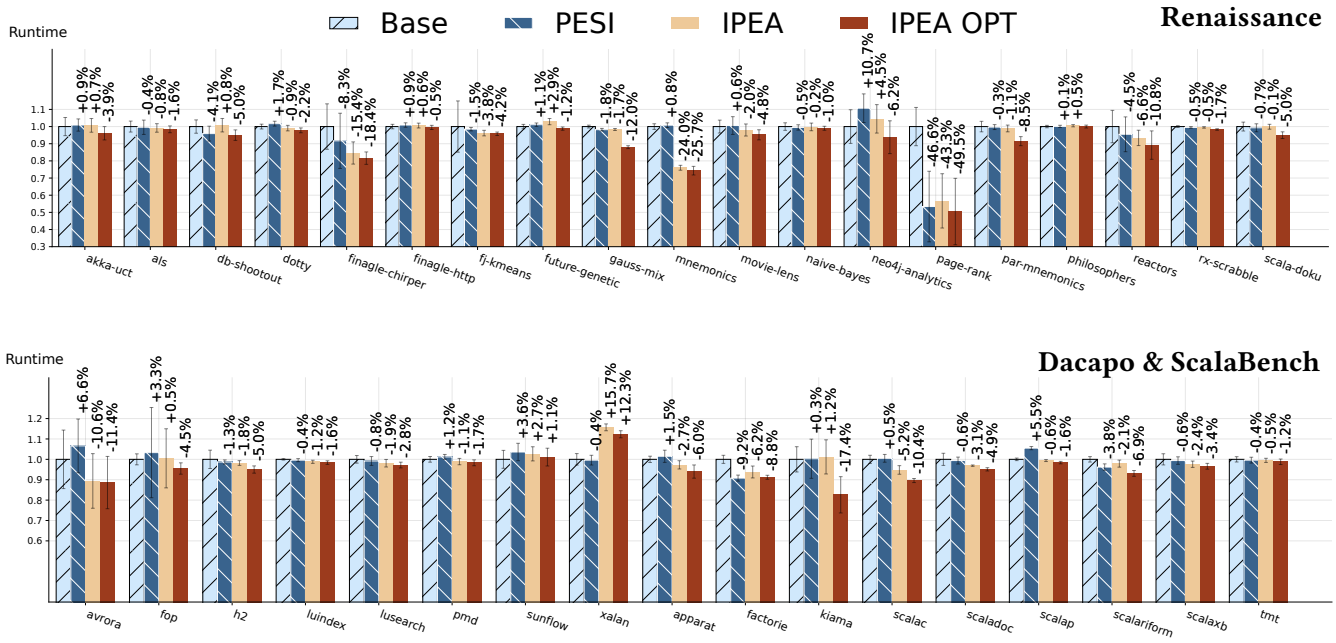


Figure 3. Relative runtime results (lower is better) in comparison to BASE of PESI, IPEA, and IPEA OPT where hyperparameters are chosen individually for each benchmark to achieve the lowest *runtime*. Some benchmarks like `mnemonics`, `finagle-chirper`, and `page-rank` reach strongly to allocation reductions, and IPEA outperforms PESI on 25 of the 36 benchmarks and significantly outperforms with an improvement of >5% on 6 benchmarks, while only having one significant regression for `xalan`.

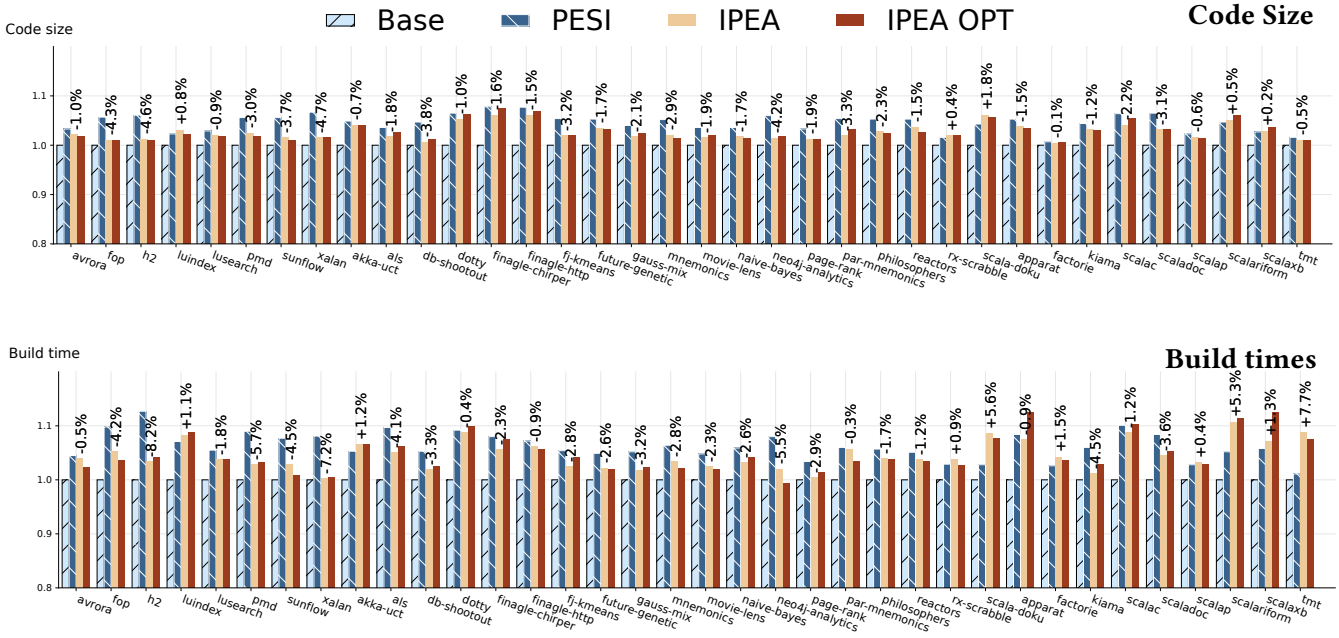
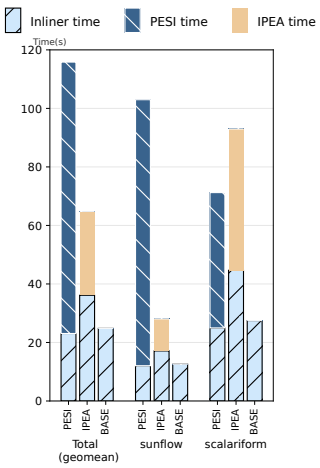


Figure 4. Build times and Code size (lower is better) of all configurations relative to BASE, where 31 of the 36 benchmarks have a lower codesize with geomean 1.81% and up to 4.98% comparing IPEA to PESI. Similarly, 27 benchmarks have lower build times with geomean improvement of 1.62% up to 8.2% faster over PESI.

Build time. Build time is kept low and shows an improvement over PESI by a geomean of 1.62% Mostly this is a result of not running IPEA for every inlining round and further throttling IPEA if procedures become increasingly large. The largest build time increases are observed for benchmarks `tmt` with +7.7% and, `scalariform` with +5.3%.

We differentiate two factors for differences in build time. First, the accumulated time spent in performing the analysis, shown in the figure on the right, and second the effects of inlining making other optimizations slower due to larger procedure bodies . Notice that IPEA is part of the incremental inliner, causing more expansion of the call tree and more inlining rounds, in addition to the time spent within IPEA itself. Nevertheless, considering both factors results in lower overall average build time.



Dynamic allocations at runtime. For more insight on the performance improvements and the effects of our analysis, we profiled the allocation sites for dynamic memory allocated at runtime, see Table 1. We stress that the number of bytes allocated is merely a fraction of the effects on runtime. Take the benchmark `scalap`. We measured higher runtimes of 5% with PESI due to code bloat by over-inlining to eliminate allocations, and therefore a higher number of instruction cache misses. We now have faster runtime, even though more bytes are allocated at runtime. Clearly, IPEA has taken other metrics into account before inlining to eliminate allocations. Concurrently, benchmarks like `mnemonics` and `finagle-chirper` seem to directly benefit from the reduction in allocations with 24.62% and 7.74% faster runtimes. However, we still see inconsistent results in some scenarios, where `factorie` has a 15% reduction in allocations, yet is 3.35% slower – allocation reduction is overvalued . We argue these results show that it is less about the number of allocations (allocations are cheap) and more about *how* these allocations interact with other elements of the code and feel our approach is a step in the right direction and can be further refined. While not displayed in this data, our anecdotal experience in analyzing the allocation sites has shown that where the allocations occur with PESI vs IPEA are quite different, even if the amount of memory allocated is similar and is worth further investigating.

Heuristic Tuning. Recall the introduction of three primary hyper-parameters that we tune for: materialization boost δ_{mb}, escape boost δ_{eb} and cutoff weight δ_{cw}. The parameters were tuned across all benchmarks in all suites. For brevity's sake, we show only a handful in Figure 5. Strong performance reactions are detected for `gauss-mix` and `kiama` where we

Table 1. Selected runtime allocation profiling results. Allocation reduction for all benchmarks can be found here [54].

	GB allocated			$\frac{IPEA-PESI}{PESI}$		
Benchmark	Base	PESI	IPEA	ΔAlloc	ΔRuntime	
avrora	2.940	1.486	2.759	+85.6%	-16.07%	Dacapo
fop	7.922	7.374	7.405	+0.41%	-2.66%	
pmd	125.3	120.4	114.0	-5.34%	-2.23%	
xalan	72.59	72.48	70.81	-2.3%	+16.17%	
geomean				+6.5%	-1.33%	
akka-uct	803.1	795.9	797.3	+0.17%	-0.23%	Renaissance
fin-chir.	1584	1527	1467	-3.91%	-7.74%	
fj-kmea.	576.4	576.2	577.2	+0.19%	-2.4%	
fut-gen.	75.34	76.19	114.2	+50.0%	+1.74%	
gauss.	110.2	108.1	108.4	+0.3%	+0.16%	
mnem.	143.7	156.5	133.4	-14.73%	-24.62%	
mo-lens	324.5	323.2	319.1	-1.26%	-2.55%	
p-rank	294.6	282.4	266.3	-5.68%	+6.13%	
geomean				-1.9%	-2.07%	
apparat	22.95	22.41	22.39	-0.09%	-4.14%	ScalaB.
factorie	116.3	106.7	90.69	-15.05%	+3.35%	
scalac	20.98	20.75	19.77	-4.73%	-5.74%	
scalap	6.664	6.493	6.573	+1.23%	-5.78%	
geomean				+2.04%	-1.61%	
total				**+1.02%**	**-1.79%**	

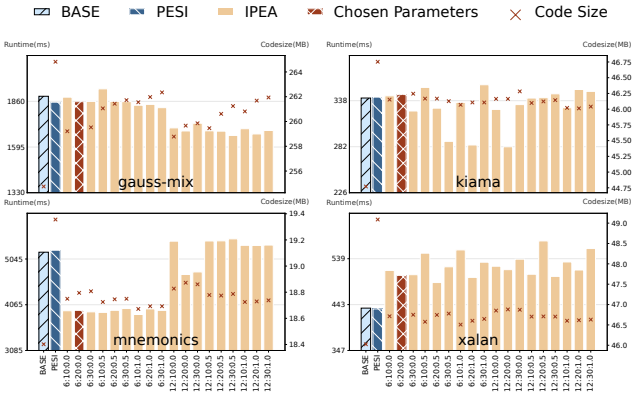


Figure 5. Selected grid search of hyper parameters, lower is better. x-labels: ($\delta_{eb} : \delta_{mb} : \delta_{cw}$). Grid search plots for entire benchmark suite can be found here [54].

were not able to extract the complete potential when optimizing for the entire suite. Our only major regression `xalan` experiences a slowdown even with a reduction in allocations. It is the only benchmark that experiences slowdowns with any parameters in our grid search. We remark that single-threaded performance of `xalan` with IPEA performs the same as PESI. Our running hypothesis is that the different inlining choices improve sequential code performance, but decrease the periods between inter-thread synchronizations,

which causes more contention. In practice, $\delta_{eb} = 6, \delta_{mb} = 20$ and $\delta_{cw} = 0$ performed the best on average. Interestingly, assigning a materialization caused by an unexplored invoke to weight 0 has the best performance. Intuitively, this helps in finding optimization opportunities for objects that get passed deep into the call graph. If an inlining decision is made based on this value, the reduction in materialization frequency is not guaranteed but the materialization is certainly delayed.

5 Related Work

It can be argued that inlining and escape analysis are the cornerstone of high-level compiler optimizations, and crucial for high-level programs and frameworks that involve (parallel) data-processing [30, 34, 38, 58], streaming [24, 29, 37], and functional programming patterns in particular [22, 28, 31, 52]. In this section, we give a brief overview of related literature ranging from escape analysis to inlining with cost-benefit estimations, and the relationship between the two.

Cost-benefit based inlining. Some of the earliest work on cost-benefit based inlining with a fixed budget is introduced by Ayers et al. [3] and shortly after by Arnold et al. [2]. While their cost model is conceptually similar to modern approaches, the benefit model used is straightforward – Ayers et al. infer the benefit based on the ability of the callee to be specialized. They use a similar approach to Ball [4] that uses data-flow analysis on the procedure arguments to predict the effects of inlining when propagating a constant to the callee. Arnold et al. use the profiled callsite frequency as the benefit metric to estimate the direct benefits of inlining by removing linkage overhead. Dean & Chambers [13] showed that the indirect benefits are equally, if not more, important than the direct benefits and perform inlining trials. The resulting optimized body is analyzed to predict benefits. With the injection of virtual state across procedure boundaries, IPEA can be seen as a sophisticated inlining trial.

Escape analysis. Choi et al. [11], Gay & Steensgaard [18], Kotzmann et al. [25] , Blanchet [8], and Whaley & Rimard. [55] all introduce a form of interprocedural escape analysis. These techniques inject states from a caller into the callee and vice versa. Both Blanchet and Kotzmann et al. suggest inlining based on the analysis results [8, 25]. Gay & Steensgaard, Whaley & Rimard, and Choi et al. use the acquired information only for stack allocation and lock elimination [11, 18, 55]. Choi et al. continue to describe a flow-sensitive interprocedural escape analysis for higher analysis accuracy, but the analysis is still only used for making binary escapability assessments [11]. Attempting to predict the frequency of dynamic allocations statically is never considered.

Profiling allocation sites. Shankar et al. [5] introduce JOLT, a profiling and inlining algorithm that targets the reduction of object churn with capture and control analysis introduced by Dufour et al. [17]. JOLT profiles object lifetimes dynamically and directly from the garbage collector. The data is used in a standalone cost-benefit inlining phase. Our evaluation showed multiple separated inlining phases can produce inconsistent results. However, combined dynamic object profiling and materialization frequency could further amplify the accuracy of the cost-benefit model.

Inlining meets Escape Analysis. Dealing with dynamic procedure calls can be challenging for program analysis [23]. Devirtualizing call sites allow further exploration of the call tree. Sewe et al. study the prediction of inlining effects on the specialization of the call tree, which in turn enables more inlining [46]. This specialization of the call tree is analogous to delaying materializations since the type of a virtual object is always known. This fact is also used by the optimization-driven incremental inliner mentioned throughout this paper by Prokopec et al. [35, 42], which runs PEA only intraprocedurally in the root procedure in between inlining rounds. This PEA runs completely disjoint from IPEA and aims to devirtualize invokes on virtual objects, and propagate type information to the callee, if a parameter is virtual at a callsite.

In a similar vein, Interflow by Shabalin et al. [48] fuses multiple techniques, including PEA, code duplication, and inlining to optimize collection-based Scala code for Scala Native. They report performing PEA in the same pass as inlining and try to always inline procedures with virtual objects at the callsite to delay the materialization. We improve on this shared concept by additionally computing materialization frequency, yet this again prompts the question – can we quantify the benefit of delaying a materialization and use value in the cost-benefit function for inlining prediction? Can we not only estimate the direct effects of inlining on allocation reduction but also encapsulate the effects of other intraprocedural optimization phases that aim to reduce runtime allocations (like code duplication)?

6 Conclusion

We presented a novel algorithm based on interprocedural partial escape analysis, which computes object-materialization frequencies and predicts the benefits of inlining. We proposed several heuristics that rely on allocation-frequency reduction to guide inlining decisions. We tuned these heuristics across 36 benchmarks from 3 benchmark suites and found that they considerably improve the runtime on 6 benchmarks, with an improvement of over 5% and up to 24.62%. In almost all cases, our IPEA-based inlining resulted in smaller image sizes compared to existing purely PEA-driven approaches. The tuning also showed that trading-off run times, compilation times and generated code-size can individually improve each of these metrics. Decent compilation overhead suggests that our algorithm can improve performance in Just-in-Time compilation with some re-tuning. Furthermore, we found compelling evidence that factors beyond allocation reduction may be used to predict performance improvements, and we believe this is a promising research direction.

References

[1] [n.d.]. V8 Engine Documentation. https://v8.dev/docs. Accessed: 2022-08-26.

[2] Matthew Arnold, Stephen Fink, Vivek Sarkar, and Peter F Sweeney. 2000. A comparative study of static and profile-based heuristics for inlining. In *Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization*. 52–64.

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

[4] J Eugene Ball. 1979. Predicting the effects of optimization on a procedure body. *ACM SIGPLAN Notices* 14, 8 (1979), 214–220.

[5] Carl E Baum, William L Baker, William D Prather, Jane M Lehr, James P O'Loughlin, DV Giri, Ian D Smith, Robert Altes, James Fockler, Donald M McLemore, et al. 2004. JOLT: A highly directive, very intensive, impulse-like radiator. *Proc. IEEE* 92, 7 (2004), 1096–1109.

[6] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. 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*. 169–190.

[7] Bruno Blanchet. 1998. Escape analysis: Correctness proof, implementation and experimental results. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 25–37.

[8] Bruno Blanchet. 1999. Escape analysis for object-oriented languages: application to Java. *Acm Sigplan Notices* 34, 10 (1999), 20–34.

[9] Bruno Blanchet. 2003. Escape analysis for JavaTM: Theory and practice. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 25, 6 (2003), 713–775.

[10] Gilad Bracha, Norman Cohen, Christian Kemper, Inprise Martin Odersky, EPFL David Stoutamire, Kresten Thorup, and Philip Wadler. 2003. Adding Generics to the Java Programming Language: Public Draft Specification Version 2.0. *Java Community Process, http://www. jcp. org/en/jsr/detail* (2003).

[11] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C Sreedhar, and Sam Midkiff. 1999. Escape analysis for Java. *Acm Sigplan Notices* 34, 10 (1999), 1–19.

[12] Jong-Deok Choi, Manish Gupta, Mauricio J Serrano, Vugranam C Sreedhar, and Samuel P Midkiff. 2003. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 25, 6 (2003), 876–910.

[13] 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*. 273–282.

[14] David Detlefs and Ole Agesen. 1999. Inlining of virtual methods. In *European Conference on Object-Oriented Programming*. Springer, 258–277.

[15] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. 2013. Graal IR: An extensible declarative intermediate representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*.

[16] Gilles Duboscq, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Speculation without regret: reducing deoptimization meta-data in the Graal compiler. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*. 187–193.

[17] Bruno Dufour, Barbara G Ryder, and Gary Sevitsky. 2007. Blended analysis for performance understanding of framework-based applications. In *Proceedings of the 2007 international symposium on Software testing and analysis*. 118–128.

[18] David Gay and Bjarne Steensgaard. 2000. Fast escape analysis and stack allocation for object-based programs. In *International Conference on Compiler Construction*. Springer, 82–93.

[19] Groovy. [n.d.]. Groovy-core/listhashmap.java at master · groovy/groovy-core. https://github.com/groovy/groovy-core/blob/master/src/main/org/codehaus/groovy/util/ListHashMap.java Accessed: 2022-09-09.

[20] Christian Häubl, Christian Wimmer, and Hanspeter Mössenböck. 2013. Context-sensitive trace inlining for Java. *Computer Languages, Systems & Structures* 39, 4 (2013), 123–141.

[21] Kim Hazelwood and David Grove. 2003. Adaptive online context-sensitive inlining. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* IEEE, 253–264.

[22] Gérard Huet. 1997. The Zipper. *J. Funct. Program.* 7, 5 (sep 1997), 549–554. https://doi.org/10.1017/S0956796897002864

[23] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. 2000. A study of devirtualization techniques for a Java Just-In-Time compiler. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 294–310.

[24] Asterios Katsifodimos and Sebastian Schelter. 2016. Apache Flink: Stream Analytics at Scale. In *2016 IEEE International Conference on Cloud Engineering Workshop (IC2EW)*. 193–193. https://doi.org/10.1109/IC2EW.2016.56

[25] Thomas Kotzmann and Hanspeter Mössenböck. 2005. Escape Analysis in the Context of Dynamic Compilation and Deoptimization. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments* (Chicago, IL, USA) *(VEE '05)*. Association for Computing Machinery, New York, NY, USA, 111–120. https://doi.org/10.1145/1064979.1064996

[26] Thomas Kotzmann and Hanspeter Mossenbock. 2007. Run-time support for optimizations based on escape analysis. In *International Symposium on Code Generation and Optimization (CGO'07)*. IEEE, 49–60.

[27] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

[28] 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

[29] Erik Meijer. 2012. Your Mouse is a Database: Web and Mobile Applications Are Increasingly Composed of Asynchronous and Realtime Streaming Services and Push Notifications. *Queue* 10, 3 (mar 2012), 20–33. https://doi.org/10.1145/2168796.2169076

[30] Erik Meijer, Brian Beckman, and Gavin Bierman. 2006. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data* (Chicago, IL, USA) *(SIGMOD '06)*. Association for Computing Machinery, New York, NY, USA, 706. https://doi.org/10.1145/1142473.1142552

[31] Erik Meijer, Maarten Fokkinga, and Ross Paterson. 1991. Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming Languages and Computer Architecture*, John Hughes (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 124–144.

[32] 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* (Monterey, California) *(JVM'01)*. USENIX Association, USA, 1.

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

[34] 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*

(Bordeaux, France) *(Euro-Par'11)*. Springer-Verlag, Berlin, Heidelberg, 136–147.

[35] Aleksandar Prokopec, Gilles Duboscq, David Leopoldseder, and Thomas Wirthinger. 2019. An Optimization-Driven Incremental Inline Substitution Algorithm for Just-in-Time Compilers. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 164–179. https://doi.org/10.1109/CGO.2019.8661171

[36] 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* (Vancouver, BC, Canada) *(SCALA 2017)*. Association for Computing Machinery, New York, NY, USA, 29–40. https://doi.org/10.1145/3136000.3136002

[37] Aleksandar Prokopec, Heather Miller, Tobias Schlatter, Philipp Haller, and Martin Odersky. 2013. FlowPools: A Lock-Free Deterministic Concurrent Dataflow Abstraction. In *Languages and Compilers for Parallel Computing*, Hironori Kasahara and Keiji Kimura (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 158–173.

[38] Aleksandar Prokopec, Dmitry Petrashko, and Martin Odersky. 2015. Efficient Lock-Free Work-Stealing Iterators for Data-Parallel Collections. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 248–252. https://doi.org/10.1109/PDP.2015.65

[39] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. On Evaluating the Renaissance Benchmarking Suite: Variety, Performance, and Complexity. *CoRR* abs/1903.10267 (2019). arXiv:1903.10267 http://arxiv.org/abs/1903.10267

[40] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomí r Bulej, Yudi Zheng, Alex Villazón, Doug Simon, et al. 2019. Renaissance: benchmarking suite for parallel applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 31–47.

[41] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomí r Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: A Modern Benchmark Suite for Parallel Applications on the JVM. In *Proceedings Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity* (Athens, Greece) *(SPLASH Companion 2019)*. Association for Computing Machinery, New York, NY, USA, 11–12. https://doi.org/10.1145/3359061.3362778

[42] Aleksandar Prokopec and Thomas Wuerthinger. 2019. Enhancing program execution using optimization-driven inlining. http://www.freepatentsonline.com/10261765.html

[43] Fabrice Rastello. 2016. *SSA-Based Compiler Design* (1st ed.). Springer Publishing Company, Incorporated.

[44] Alexandru Salcianu and Martin Rinard. 2001. Pointer and escape analysis for multithreaded programs. *ACM SIGPLAN Notices* 36, 7 (2001), 12–23.

[45] Robert W Scheifler. 1977. An analysis of inline substitution for a structured programming language. *Commun. ACM* 20, 9 (1977), 647–654.

[46] 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*. 317–328.

[47] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. 2011. Da Capo Con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 657–676. https://doi.org/10.1145/2048066.2048118

[48] Denys Shabalin and Martin Odersky. 2018. Interflow: interprocedural flow-sensitive type inference and method duplication. In *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala*. 61–71.

[49] M Šipek, D Muharemagić, B Mihaljević, and A Radovan. 2020. Enhancing Performance of Cloud-based Software Applications with GraalVM and Quarkus. In *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)*. IEEE, 1746–1751.

[50] 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* (Orlando, FL, USA) *(CGO '14)*. Association for Computing Machinery, New York, NY, USA, 165–174. https://doi.org/10.1145/2581122.2544157

[51] 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*. 221–226.

[52] Philip Wadler. 1993. Monads for functional programming. In *Program Design Calculi*, Manfred Broy (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 233–264.

[53] Sutao Wang. 2021. *Thin Serverless Functions with GraalVM Native Image*. Master's thesis. ETH Zurich.

[54] Matthew Edwin Weingarten, Theodoros Theodoridis, and Aleksandar Prokopec. 2022. Inlining-Benefit Prediction with Interprocedural Partial Escape Analysis – Appendix. https://doi.org/10.5281/zenodo.7308640

[55] John Whaley and Martin Rinard. 1999. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 187–206.

[56] Christian Wimmer, Vojin Jovanovic, Erik Eckstein, and Thomas Würthinger. 2017. One Compiler: Deoptimization to Optimized Code. In *Proceedings of the 26th International Conference on Compiler Construction* (Austin, TX, USA) *(CC 2017)*. Association for Computing Machinery, New York, NY, USA, 55–64. https://doi.org/10.1145/3033019.3033025

[57] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. 2019. Initialize Once, Start Fast: Application Initialization at Build Time. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 184 (oct 2019), 29 pages. https://doi.org/10.1145/3360610

[58] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (oct 2016), 56–65. https://doi.org/10.1145/2934664

Received 2022-09-18; accepted 2022-10-05