U Can't Inline This!

Erick Ochoa*
SBA Research
eochoalopez@sba-research.org

Cijie Xia†
University of Toronto
cijie.xia@mail.utoronto.ca

Karim Ali
University of Alberta
karim.ali@ualberta.ca

Andrew Craik‡
Oracle Labs
andrew.craik@oracle.com

José Nelson Amaral
University of Alberta
jamaral@ualberta.ca

ABSTRACT

Inlining is a compile-time program transformation that reduces the overhead of method calls and enables further program transformations. Within a Java just-in-time (JIT) compiler, an inlining algorithm mostly relies on a mixture of hard-coded heuristics and call-frequency information to decide which functions to inline. To operate on those heuristics, a traditional Java JIT inliner employs a greedy knapsack algorithm that trades off computing an optimal solution for the algorithm runtime performance. However, maintaining those hard-coded heuristics is a difficult, time-consuming task that requires years of domain expertise to fine tune. To overcome those limitations, we present OurInliner, a Java JIT inlining framework that uses abstract interpretation to systematically evaluate inlining candidates based on both direct and indirect benefits of inlining. To reduce the compilation overhead in the context of a Java JIT compiler, OurInliner computes reusable method summaries. Each summary defines the potential optimizations that will be unlocked if the Java JIT compiler inlines the underlying method (i.e., indirect benefits). Similar to prior work, OurInliner incorporates call-frequency information (i.e., direct benefits) to compute the benefit of inlining and defines the cost of inlining a function as its code size. To showcase the viability of our framework, we have implemented OurInliner for the open-source Eclipse OpenJ9 Java virtual machine. Compared to the state-of-the-art JIT inliner in Eclipse OpenJ9, our empirical evaluation shows that OurInliner reduces compilation time by 14% and decreases the size of generated code by 21%, while incurring a run-time overhead of only 5%.

ACM Reference Format:
Erick Ochoa, Cijie Xia, Karim Ali, Andrew Craik, and José Nelson Amaral. 2021. U Can't Inline This!. In *Proceedings of (CASCON'21).* ACM, New York, NY, USA, 10 pages.

1 INTRODUCTION

Inlining is a program transformation that replaces a call site with the body of the callee. This transformation has two main benefits: (1) it eliminates the overhead of method invocation and frame allocation costs, and (2) it allows the compiler to further optimize the code of the inlined function into its calling method. However, function inlining is a program transformation that must be applied selectively. Indiscriminate inlining leads to an increase in binary size, compilation time, cost in program storage/transmission, and potentially negative cache effects that diminish much of the benefits of inlining. To select candidate functions, an inlining strategy must carefully balance the trade-off between costs and benefits [15]. Such strategy usually consists of four main components:

(1) *Discriminants*: the properties that influence inlining.
(2) *Inlining budget*: the inlining constraints that avoid an uncontrollable increase in binary size.
(3) *Search space*: the limited set of inlining candidates.
(4) *Algorithm*: determining the order in which inlining candidates should be considered and whether inlining decisions may be recanted after finding new inlining candidates.

Early work on inlining strategies uses method sizes and frequency information (in the form of method invocation counters) as discriminants [21]. More recent research focuses on providing more precise frequency information (e.g., context-sensitive call site counters) [4, 14] or combine other sources of frequency information (e.g., cycle counts) into other metrics (e.g., cycle density) [25]. However, collecting online frequency information may degrade performance and is only useful if profiled behaviour is indicative of future program behaviour. To improve the precision of inlining decisions, compiler engineers may also consider static program properties such as polymorphic call sites, interface call sites, and the function fan-out degree. Several inlining strategies also discriminate based on which optimizations will be unlocked after inlining [19, 22, 23]. Nevertheless, those techniques predominantly depend on determining the algorithm parameters empirically through inlining trials [11], which is a difficult, time-consuming task that requires years of domain expertise to fine tune [7].

To overcome those limitations, we present OurInliner, a Java JIT inlining framework that uses abstract interpretation to systematically evaluate inlining candidates without performing inlining trials or fine-tuning the algorithm parameters by domain experts. Similar to prior work, OurInliner defines the cost of inlining a function to be its code size. To compute the benefit of inlining, OurInliner combines the typical call-frequency information (i.e., direct benefit) with the potential optimizations that may be unlocked after inlining (i.e., indirect benefit). To reduce the compilation overhead, OurInliner uses abstract interpretation to compute reusable inlining summaries that contain predicates relating potential runtime values of method arguments to potential post-inlining optimizations. To define the search space for inlining candidates, our framework employs the notion of Inlining Dependency Tree (IDT) [10], which is an abstract representation of the Control-Flow Graph (CFG) of the program under analysis. Our framework then formulates function inlining as a nested knapsack problem [10]. Unlike traditional knapsack formulations of inlining [4, 14, 21–25],

*The work was done while the author was at the University of Alberta.
†The work was done while the author was at the University of Alberta.
‡The work was done while the author was at IBM Canada.

CASCON'21, November 22 - 26 2021, Toronto, Canada

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.

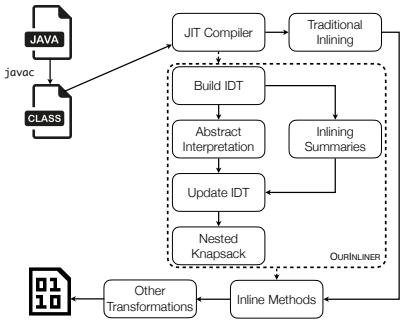


Figure 1: The Java JIT compilation process. The dotted region represents the new components of OurInliner.

the nested knapsack formulation enables backtracking across many levels along the IDT if it later finds better inlining candidates.

We demonstrate the viability of OurInliner through a prototype implementation for the Eclipse OpenJ9 Java Virtual Machine (JVM) [13]. We show that, across the Java DaCapo benchmarking suite [6], OurInliner reduces JIT compilation time by 14% and the generated code size by 21%, while incurring a run-time overhead of only 5% compared to the stock Eclipse OpenJ9 JIT inliner.

2 OVERVIEW OF OURINLINER

Inlining is a code transformation placed early in the compilation flow because it may enable further optimizations. Figure 1 illustrates the JIT compilation process of a given Java source file, including where inlining takes place. At runtime, the JVM may issue a compilation request for a specific method. The JIT compiler applies all transformations in the compilation plan that corresponds to that request. This step produces a compiled representation of the method for which the JVM has issued the compilation request. The JVM then substitutes the uses of the original bytecode method representation with the newly compiled method representation.

OurInliner replaces the traditional inlining transformation pass shown in Figure 1 with a new set of components (shown within the dotted region). First, OurInliner defines the search space for inlining using the IDT data structure. OurInliner then computes reusable method summaries that determine the invariants that hold at a given call site for an optimization to take place. To compute the invariants at that specific call site, OurInliner performs abstract interpretation on the bytecode instructions of the method body. To update the IDT with a notion of benefit, OurInliner applies the computed method summaries to the static information that the abstract interpreter has found. Based on the results computed at this step, our nested knapsack algorithm selects the functions (i.e., nodes of the IDT) that should be inlined.

3 BUILDING AN INLINING DEPENDENCY TREE

Figure 1 shows that the first step in OurInliner is building an IDT, which is an abstract representation of the CFG of the program that models inlining decisions. Building an IDT is a recursive process where the stopping condition depends on not finding call sites or call targets or the call sites and call targets that OurInliner finds do not fit within the remaining inlining budget.

Initially, the IDT contains only the root node (i.e., the method requested for compilation). Each node in the IDT is annotated with its inlining budget (i.e., the cost that corresponds to the method size). Therefore, the budget of the root node corresponds to the total inlining budget allowed by the JVM for this specific compilation request. To build the IDT, OurInliner inspects the bytecode corresponding to the root node for inlining candidates. When OurInliner finds an inlining candidate whose size is less than the current budget, it creates a new IDT node corresponding to the inlining candidate and adds an edge between that node and the root node. OurInliner then computes the budget of the new node by subtracting the size of the inlining candidate from the budget of the root and associates it to the newly created node. After OurInliner has built an IDT for one of the call sites that it found in a method, it continues inspecting the rest of the bytecodes for more call sites.

3.1 Example of Building an IDT

Figure 2 shows an example Java program and the IDT that OurInliner builds for it. Let us assume that the JVM issues a compilation request for the root method `Example.main()`. To build the IDT, OurInliner iterates over the bytecode instructions of `Example.main()` looking for call sites. When OurInliner finds a call site, it first determines its type: static, virtual, interface, or dynamic. For the static call to `Base.staticFun()` (Line 3), OurInliner adds a node to the IDT that corresponds to the only target for this call site. To distinguish between different call sites, OurInliner annotates the edge between the root node and the node containing the call target `Base.staticFun()` with the bytecode at which it found the call site. OurInliner then inspects the body of `Base.staticFun()`, but it does not find more call sites. Therefore, it returns to Line 4 to continue inspecting the body of the root method `Example.main()`.

For the virtual call at Line 4, OurInliner must distinguish between the various potential receivers of the call. Therefore, it adds two nodes to the IDT for `Base.virtualSingle()` and `Derived.virtualSingle()`. Naturally, for the virtual call site with multiple implementors at Line 5, OurInliner should maintain context-sensitivity. Therefore, it adds two nodes to the IDT, one for `Base.virtualMultiple()` and another for `Derived.virtualMultiple()`.

Finally, OurInliner finds two call sites to the method `Base.ctx()` (Lines 6 and 7). For each call site, OurInliner creates a node in the IDT and annotates the edge between that node and the root node with the bytecode index of the call site. Depending on how polymorphic a call site is, the implementation of OurInliner is flexible enough to limit how many nodes it creates by only adding the most likely target of a virtual call making use of the runtime information that is available through the underlying JIT compiler.

3.2 Computing Direct Benefits of Inlining

Given that OurInliner constructs the IDT at run time, it uses two sources of JIT profiling information: (1) *Basic-block frequency:* the proportion of time likely spent on a basic block for each method, and (2) *Target frequency:* the percentage of calls to each target of a specific call site, represented as a value in the range [0–1]. To model the direct benefits of inlining, OurInliner uses this profiling information to compute the call ratio, which estimates how many times OurInliner visits an IDT child node for each execution of its parent. OurInliner then annotates each node in the IDT with its call ratio. By definition, the call ratio for the root node (`Example.main()` in Figure 2) is 1. The following equation calculates

```
1  class Example {
2    public static void main(Base b) {
3      Base.staticFun();
4      b.virtualSingle();
5      b.virtualMultiple();
6      for (int i = 0; i < 10; i++) { Base.ctx(); }
7      if (p1 < 0.5) { Base.ctx(); }
8    }
9  }
10 class Base {
11   public static void staticFun() { }
12   public static void ctx() { if (p2 < 0.5) { Base.staticFun(); } }
13   public void virtualSingle() { }
14   public void virtualMultiple() { }
15 }
16 class Derived extends Base {
17   public void virtualMultiple() { }
18 }
```

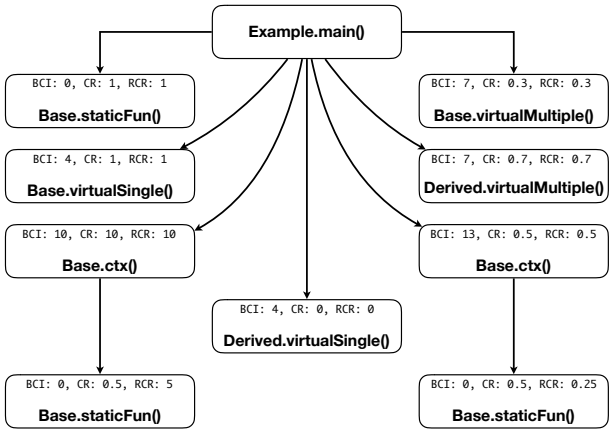


Figure 2: Left: a Java program that illustrates how OurInliner constructs the IDT. In this example, `p1` and `p2` represent stochastic random variables of a uniform distribution. Right: the corresponding IDT with bytecode indices as edge annotations to distinguish multiple calls to the same method. BCI: bytecode index, CR: call ratio, and RCR: root call ratio.

the call ratio for node n in the IDT:

$$\textit{call ratio}(n) = \frac{\textit{basic-block frequency at call site}}{\textit{basic-block frequency at parent entry}} \times \textit{target frequency}$$

To estimate how many times an IDT node is visited for each execution of the root node, OurInliner computes the *root call ratio* for a given node as the product of the call ratios of its ancestors:

$$\textit{root call ratio}(n) = \prod_{\forall m \in \textit{ancestors}(n) \cup n} \textit{call ratio}(m)$$

3.3 Example of Computing Call Ratios for an IDT

This example illustrates the computation of the call ratios for the IDT nodes for the call target `Base.ctx()` shown in Figure 2. Each call to `Example.main()` calls `Base.ctx()` at Line 6 ten times. Therefore, OurInliner annotates the corresponding node with the call ratio 10. Each call to `Example.main()` calls `Base.ctx()` at Line 7 only half the time. Therefore, OurInliner annotates the corresponding node with the call ratio of 0.5. The processing of the bytecodes of `Base.ctx()` reveals that each call to the method calls `Basic.staticFun()` at Line 12 only half the time. Therefore, OurInliner annotates the corresponding node with a call ratio of 0.5. However, the root call ratio for each node that corresponds to the call site at Line 12 is distinct with respect to its ancestor. For the node whose parent is the call site at Line 6, the root call ratio is 5. For the node whose parent is the call site at Line 7, the root call ratio is 0.25.

4 ABSTRACTING THE JVM STATE

To model the JVM state at specific program points, OurInliner employs abstract interpretation [9]. The reader might wonder why we need this abstraction if we are already at run time and know exactly what the JVM state is. The main reason is that OurInliner captures optimizations that may occur without any optimistic assumptions about the code under analysis, which is key to reduce potential performance overhead for the underlying Java JIT compiler. Therefore, OurInliner determines a safe state approximation at points just before it finds a call site. Designing the abstract interpretation in OurInliner boils down to two main design decisions.

First, how much of the JVM state should OurInliner model? Given the JIT context of OurInliner, it is prudent to model the least amount of information while still producing beneficial results. To capture enough of the JVM state, OurInliner models the local variable array, the operand stack, and the call stack. To reduce the memory footprint of OurInliner, it currently does not model the heap. Once values enter the heap, OurInliner loses all its information about the abstract values. Upon retrieving values from the heap, OurInliner only considers the bytecode instruction itself.

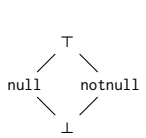
Second, what is a precise abstraction for each concrete value that OurInliner may encounter? To address this concern, OurInliner places abstract values from the following lattices on the *abstract operand stack* or the *abstract local variable array*:

(1) a boolean lattice to model `boolean` primitive types,
(2) a long-range lattice to model `long`-like primitive types (i.e., short, integer, and long),
(3) the $\top$ singleton lattice to model `float` and `double`, and
(4) a composed lattice [16] to model references, which is the product of a boolean lattice to model `null`-ness, a long-range lattice to model array lengths, and a class-hierarchy lattice to model classes.

Figure 3 depicts the lattices that OurInliner uses. The merge operation for the boolean and long-range lattices is the standard merge function. For the class-hierarchy lattice, the merge function takes two classes and yields the first common ancestor in the class hierarchy. For the composed lattice for references, OurInliner performs the merge function individually on each of its components. Figure 3 shows the merge function for the `null` lattice, examples of merging long ranges for the range lattice and of merging references for the class-hierarchy lattice. The merge operations for the abstract operand stack and the abstract local variable array work by iterating the individual abstract values and merging them in order.

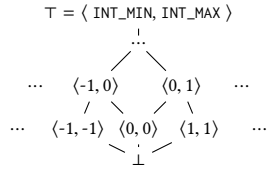
4.1 Abstract Interpretation in a JIT Compiler

The decision to not track abstract values once they are stored on the heap impacts how OurInliner abstracts several JVM bytecode instructions. In particular, the instructions `getfield` and `getstatic` on non-primitive data types should always return the abstract tuple



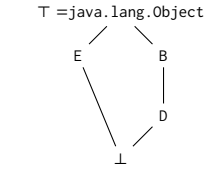
	⊤	null	notnull
⊤	⊤	⊤	⊤
null	⊤	null	⊤
notnull	⊤	⊤	notnull

(a) The `null`-ness lattice.



	⊤	⟨0, 0⟩	⟨-10, 10⟩	⟨1, 4⟩
⊤	⊤	⊤	⊤	⊤
⟨0, 0⟩	⊤	⟨0, 0⟩	⟨-10, 10⟩	⟨0, 4⟩
⟨-10, 10⟩	⊤	⟨-10, 10⟩	⟨-10, 10⟩	⟨-10, 10⟩
⟨1, 4⟩	⊤	⟨0, 4⟩	⟨-10, 10⟩	⟨1, 4⟩

(b) The long-range lattice.



	⊤	E	B	D
⊤	⊤	⊤	⊤	⊤
E	⊤	E	⊤	⊤
B	⊤	⊤	B	B
D	⊤	⊤	B	D

(c) The class-hierarchy lattice.

Figure 3: The lattices, and their merge functions, that OurInliner uses to abstract concrete values in the JVM.

⟨⊤, ⊤, ⊤⟩, indicating that OurInliner does not know whether the value may be `null`, whether it is an array reference, or its class type. To obtain a more precise estimate, OurInliner uses the information in the constant pool of the Java classe files that are currently running. The constant pool records the base class of the field that OurInliner obtains through `getfield`, thus OurInliner computes the more precise estimate ⟨⊤, ⊤, `ClassName`⟩, indicating that OurInliner now knows its class type. OurInliner models other instructions that obtain values from the heap such as getting values from arrays in a similar fashion.

Another consideration for doing abstract interpretation within a JIT compiler is that OurInliner cannot feasibly perform whole-program analysis. Therefore, the abstract interpretation is restricted to methods that are in the IDT. As a result, if OurInliner finds a call site without a corresponding IDT node, it conservatively abstracts the return with the one given by the type signature. In the current implementation, static information only flows down the IDT (i.e., from call sites). However, future extensions of this implementation may propagate information up the IDT (i.e., from return sites). To perform abstract interpretation efficiently in the JIT context, OurInliner traverses the basic blocks of a given method in reverse post-order. This order ensures that, in the absence of cycles, OurInliner would have interpreted all the predecessors of a node before interpreting the node itself.

To model how the JVM maintains the concrete counterparts of the abstract operand stack, the abstract variable array, and the abstract call stack, we have defined the abstract semantics of all JVM bytecode instructions in OurInliner. At the beginning of the interpretation of a basic block x, OurInliner transfers the abstract state from the direct predecessors of x (according to the control-flow graph) to x. At the end of the interpretation of x, OurInliner stores the abstract state on the basic blocks and uses the abstract state as an input to the successors of x. To compute the next abstract state, OurInliner uses the incoming abstract state and applies the abstract semantics of the current instruction to it.

To facilitate abstract interpretation, $directPredecessors(x)$ finds the direct predecessors of a basic block x in the CFG and $backEdges(x)$ finds the back edges to it. OurInliner thus handles three cases:

Case 1 $|directPredecessors(x)| = 1$ and $directPredecessor(x) = y$: y has already been interpreted. Therefore, the final abstract state in y is the input to the interpretation of x.

```
public static void example(int);
  0: iconst_0
  1: istore_1
  2: iload_0
  3: ifge 11
  6: iconst_1
  7: istore_1
  8: goto 14
 11: bipush 100
 13: istore_1
 14: new D
 17: dup
 18: D."<init>":()V
 21: astore_2
 22: aload_2
 23: iload_1
 24: iload_0
 25: foo:(LB;II)V
 28: return
```

Figure 4: An example illustrating how OurInliner performs abstract interpretation.

Case 2 $|directPredecessors(x)| > 1$, $|backEdges(x)| = 0$, and $directPredecessors(x) = Y$: all nodes in Y have been interpreted, and thus the input to the interpretation of x is the merge of their final abstract states.

Case 3 $|directPredecessors(x)| > 1$, $|backEdges(x)| > 0$, and $backEdges(x) = Z$: the direct predecessor of x and its final abstract state are known. However, the abstract states of the nodes along the back edges to x are unknown. Therefore, to compute a safe approximation, OurInliner widens those abstract values to ⊤ right before interpreting x.

4.2 Example of Abstract Interpretation

In Figure 4, to estimate the arguments to the `invokestatic` instruction at bytecode index 25, OurInliner starts interpretation at bytecode index 0. At that point, OurInliner does not have any information about the variable located at index 0 in the local variable array, except that it is an integer. At bytecode index 3, the branch condition depends on the argument to `example()`. Since OurInliner does not know what values the argument may take, it analyzes all possible branches, copying the abstract state at each control-flow branch. At each bytecode index, Table 1 reflects the changes to the abstract variable array. At bytecode index 8, OurInliner creates the abstract value ⟨1, 1⟩ to model the constant 1. OurInliner modifies another copy of the state to reflect the bytecodes at indices 11 and 13. Given the merge point at bytecode index 14, OurInliner merges the states corresponding to each branch to compute the abstract value $\langle 1, 1\rangle \sqcup \langle 100, 100\rangle = \langle 1, 100\rangle$. Finally, at bytecode index 25, the abstract operand stack and abstract variable array contain the final approximations that OurInliner has computed for each local variable in the method `example()`.

Table 1: The values in the abstract local variable array (LVA) for the bytecode indices of the example in Figure 4.

Bytecode Index	LVA Index	Abstract Value
3	0	$\top$
8	0	$\top$
8	1	$\langle 1, 1 \rangle$
13	0	$\top$
13	1	$\langle 100, 100 \rangle$
14	0	$\top$
14	1	$\langle 1, 100 \rangle$
25	0	$\top$
25	1	$\langle 1, 100 \rangle$
25	2	$\langle$notnull, $\bot$, D$\rangle$

5 HOW DOES OURINLINER HANDLE LOOPS?

Upon entering a loop, OurInliner has so far used the least precise abstraction of the JVM state (i.e., $\top$). To handle loops more precisely, OurInliner should use an iterative dataflow algorithm that applies successive passes of abstract interpretation to a program until the program state converges to a stable value [18]. To reduce the run-time overhead of this iterative solution in the JIT context, OurInliner uses a structural analysis and a dependency analysis. These two analyses help the abstract interpreter in OurInliner determine whether the abstract JVM state may be affected by future iterations of the looping construct under analysis.

5.1 Structural Analysis

To identify the basic blocks of a loop, we employ a structural analysis that produces a set of regions. Each region contains a set of basic blocks or other regions. The region header dominates all basic blocks in it [18]. Our analysis identifies regions that cannot be reduced to a single control-flow node (i.e., improper regions), regions that contain a cycle (i.e., loops), and regions with no cycles. To properly handle regions that contain other regions, the abstract interpreter in OurInliner treats a region as a reduced CFG. Thus, the abstract interpretation of a region recursively interprets nested regions and blocks in reverse post-order traversal. Performing abstract interpretation on an improper region yields the least precise abstract representation of the JVM state ($\top$). By definition, the smallest region that contains two regions r_0 and r_1 is their common ancestor, which OurInliner computes via the helper function $commonAncestor(r_0, r_1)$.

5.2 Dependency Analysis

To identify values in the abstract state that depend on previous iterations of a loop, we designed a dependency analysis that reduces the number of abstract-interpretation passes required to handle loops. To support this analysis, we augment the representation of an abstract value $\hat{a}$ to include (1) d: a set of other abstract values that $\hat{a}$ depends on and (2) r: the region that $\hat{a}$ originates from. Given an abstract value $\hat{a}$, the dependency analysis computes the augmented abstract value $\hat{a}' = \langle \hat{a}, d, r \rangle$. To compute each of these components, we have defined the helper functions $value(\hat{a}')$, $dependencies(\hat{a}')$, and $region(\hat{a}')$. To support the computation of these functions, we have extended all operations of the abstract interpreter. In particular, if an interpretation produces a new value $\hat{a}'_{new}$, the dependency set of $\hat{a}'_{new}$ includes all values used in the interpretation as well as the union of the dependency sets of these values: $dependencies(\hat{a}'_{new}) = \bar{A} \cup \bigcup_{\hat{a}'_i \in \bar{A}} dependencies(\hat{a}'_i)$ where $\bar{A}$ is the set of all abstract values used in the abstract interpretation that produces the new abstract value $\hat{a}'_{new}$. For example, for an addition operation between two abstract integer values, the set $\bar{A}$ contains the two abstract integer operands. Operations that move values to different locations in the state (e.g., pop) do not create new values. Operations without operands create new values that have an empty set of dependencies. Similarly, merge functions between abstract values (Figure 3) record the dependencies of new abstract values. Given the abstract values $\hat{a}'_0 = \langle \hat{a}_0, d_0, r_0 \rangle$ and $\hat{a}'_1 = \langle \hat{a}_1, d_1, r_0 \rangle$, the merged abstract value is $\hat{a}'_0 \sqcap \hat{a}'_1 = \langle \hat{a}_0 \sqcap \hat{a}_1, d_0 \cup d_1 \cup \{\hat{a}'_0, \hat{a}'_1\}, r_0) \rangle$.

5.3 Abstract Interpretation of Loops

The structural analysis in OurInliner determines the merge block of a loop in the CFG that has an incoming back edge. When OurInliner first encounters this merge block during abstract interpretation, it has not yet interpreted its predecessor blocks that are connected through back edges. Therefore, OurInliner merges a special abstract state $\hat{B} = \langle \bot, b, r \rangle$ from the back edge and then performs a single pass through the enclosing region of the loop (i.e., r). Once OurInliner has interpreted the block that is at the origin of a back edge, it updates the set of dependencies b. If the new value for b is different from the original one when OurInliner first encountered the back edge, OurInliner merges both values together and sets r as the current region (i.e., *currentRegion*) for subsequent analysis queries that OurInliner may issue.

After this pass, the abstract values obtained by $value(\hat{a}')$ may be imprecise because OurInliner has not modelled all values that may appear in future loop iterations. To get a better approximation for these values, we define $query(\hat{a}', currentRegion)$ that OurInliner invokes when it needs an approximation of a value at a given instruction in the code under analysis. To run these queries, we define the helper function $moreIterations(\hat{a}') = |\forall \hat{a}'_d \in dependencies(\hat{a}') \,.\, \{commonAncestor(currentRegion, region(\hat{a}'_d))\} \cup \{region(\hat{a}')\}| > 1$, where the call to *commonAncestor*() helps OurInliner find loop-invariant values. To achieve that, OurInliner treats the values that originate from regions nested in *currentRegion* as if they originate from *currentRegion*. The abstract interpreter does not re-consider those nested regions, because it has already exited them. Each invocation of *query*() results in one of the following cases:

(1) If an augmented value depends on itself (i.e., $\hat{a}' \in dependencies(\hat{a}')$), then OurInliner widens its abstraction to $\top$.
(2) If $moreIterations(\hat{a}')$ returns true, then $value(\hat{a}')$ requires successive passes of abstract interpretation to obtain a better approximation for the augmented abstract values. This case occurs when a value $\hat{a}'$ depends on multiple regions. OurInliner may then narrow $\hat{a}'$ by performing further passes of abstract interpretation on the largest region in the set computed by $moreIterations(\hat{a}')$. Similar to a traditional fix-point solution for dataflow problems, OurInliner stops iterating when all dependencies of $\hat{a}'$ stop changing, because no other value may affect $\hat{a}'$.
(3) If $moreIterations(\hat{a}')$ returns false, then OurInliner does not perform further passes of abstract interpretation to find

```
38 int a, b, c, d, e; // method parameters
39 while (cond) {
40     c = b + 1;
41     b = 42;
42     e = d;
43     d = d + 1;
44 }
```

Figure 5: An example illustrating the three cases that OurInliner handles for loops. Variables `d` and `e` are examples of the first case, `b` and `c` are examples of the second case, and `a` is an example of the third case.

a better approximation for the values stored in $value(\hat{a}')$. This case occurs when these values are loop-invariant.

The combination of the structural and the dependency analyses leads to a reasonable approximation after modelling a single iteration through a loop. However, OurInliner may be configured to model further iterations of the loop by running more passes of the combination of both analyses. This flexibility allows a JIT compiler to configure OurInliner to better suit its needs.

Figure 5 shows an example where the parameters originate from region r_0 and values created inside the loop originate from region r_1 such that $commonAncestor(r_0, r_1) = r_0$. Initially, OurInliner abstracts each parameter with value $\hat{a}'_l = \langle \hat{a}_l, \emptyset, r_0 \rangle$ where l stands for the variable name. The following is a step by step abstract interpretation of the first pass on that code snippet:

$$\begin{aligned}
\hat{a}'_{c_{40}} &= \hat{a}'_b + 1 = \langle \hat{a}_b + 1, \{\hat{a}'_b, b_c\}, r_1 \rangle \\
\hat{a}'_{b_{41}} &= 42 \qquad = \langle \langle 42, 42 \rangle, \emptyset, r_1 \rangle \\
\hat{a}'_{e_{42}} &= \hat{a}'_d \qquad = \langle \hat{a}_d, \{b_d\}, r_0 \rangle \\
\hat{a}'_{d_{43}} &= \hat{a}'_d + 1 = \langle \hat{a}_d + 1, \{\hat{a}'_d, b_d\}, r_1 \rangle
\end{aligned}$$

After the first pass of abstract interpretation on this loop, OurInliner updates the back-edge dependencies accordingly. OurInliner then finds that it should apply more iterations of abstract interpretation to find more precise values for the variables b and c, which yields the following abstractions:

$$\begin{aligned}
\hat{a}'_{b_{40}} &= \langle \hat{a}_b \mid 42, \{\hat{a}'_b, b_c\}, r_1 \rangle \\
\hat{a}'_{c_{40}} &= \langle \hat{a}_b + 1 \mid 43, \{\hat{a}'_b, b_c\}, r_1 \rangle
\end{aligned}$$

6 ESTIMATING BENEFITS OF POTENTIAL POST-INLINING TRANSFORMATIONS

In static analysis, a method summary relates dataflow facts at the method entry to those at the method exit [20]. In OurInliner, *inlining summaries* relates dataflow facts at the method entry with predicates that test these facts against a specific constraint. OurInliner uses those predicates to compute the indirect benefits of inlining (i.e., attributed to code transformations enabled by inlining). For example, if a method has a branch that depends on an argument, an optimizer may fold the branch if it statically determines that the argument is constant. In an inlining summary, each predicate has the form $f_m(\hat{a}_n) = \hat{a}_n \sqsubseteq \hat{c}_m$, where f_m indicates testing m in the inlining summary, $\hat{a}_n$ indicates testing for the n-th argument abstraction, and $\hat{c}_m$ is the constraint associated with test m. An inlining summary may hold zero or more m predicates. Arguments $\hat{a}_n$ will be tested against $\hat{c}_m$. The test always checks whether two elements conform to the partial-order relationship $\hat{a}_n \sqsubseteq \hat{c}_m$. $\hat{c}_m$ corresponds to the maximal safe value where the transformation is possible. Therefore, $\hat{a}_n \sqsubseteq \hat{c}_m$ may unlock a post-inlining optimization. To compute a single numeric value for the indirect benefit of inlining, OurInliner assigns a weight w_m to each predicate f_m in an inlining summary. Then, the sum of all weighted predicates for a specific call site is the indirect benefit of inlining that call (i.e., $\sum_{i=0}^{m} f_i(\hat{a}_n) \times w_i$). OurInliner then uses this value along with the root call ratio to solve the nested knapsack problem in the IDT, which eventually determines an optimal set of functions to inline.

6.1 Determining Predicate Constraints

OurInliner enables a compiler engineer to compute the constraint c_m through their custom static analyses. OurInliner then uses the results of these analyses to generate the inlining summaries. The current implementation of OurInliner computes constraints based on a set of static analyses that reason about: `null` checking, branch folding, `cast` folding, and `instanceof` folding. To compute the constraints, the abstract interpreter in OurInliner determines whether function arguments have been modified and tracks the uses of arguments within a single method. Therefore, this interpreter models values using a boolean lattice that states whether a value has been modified or not.

To incur minimal overhead to the running JIT compiler, the abstract interpreter processes one method at a time by abstracting only the local variable array and the operand stack. The JVM Specification [17] states that upon entry to a function, the arguments are stored in the local variable array. To model this behaviour, our abstract interpreter places unmodified abstract values in the abstract variable array. For bytecode instructions that modify these abstract values, the abstract interpreter models their side effects to compute the modified abstract value. When the abstract interpreter encounters an instruction that operates on an unmodified abstract value, and that instruction can be folded away depending on that abstract value, OurInliner creates a predicate in the inlining summary.

6.2 Supported Analyses

6.2.1 Branch Folding. Figure 6a shows an example where a branch fold depends on the value of the only argument to a method. To construct the inlining summary, OurInliner first places an unmodified abstract value on the abstract variable array at bytecode index 0 to emulate entry to `branch()`. It then loads this argument onto the abstract operand stack, which is later used by `ifeq` at bytecode index 1. Given the JVM semantics of `ifeq`, if the argument is 0, then the control flow jumps to bytecode index 6. Otherwise, the control flow falls through to bytecode index 4. To model these execution steps, OurInliner generates the following inlining summary:

$$\begin{aligned}
f_0(\hat{a}_0) &= \hat{a}_0 \sqsubseteq \langle \texttt{INT_MIN}, \texttt{-1} \rangle \\
f_1(\hat{a}_0) &= \hat{a}_0 \sqsubseteq \langle 0, 0 \rangle \\
f_2(\hat{a}_0) &= \hat{a}_0 \sqsubseteq \langle 1, \texttt{INT_MAX} \rangle
\end{aligned}$$

6.2.2 `Null`*-Check Folding.* Calls to `java.lang.Object.getClass()` are used by some programmers as an implicit `null` check. In Figure 6b, an unmodified abstract value corresponding to the first

```
boolean branch(boolean);
  0: iload_0
  1: ifeq 6
  4: iconst_1
  5: ireturn
  6: iconst_0
  7: ireturn
```
(a) Branch folding

```
void nullCheck(Example);
  0: aload_0
  1: Object.getClass()
  4: pop
  5: return
```
(b) Null-check folding

```
void checkCast(Base);
  0: aload_0
  1: checkcast Derived
  4: astore_1
  5: return
```
(c) Checkcast folding

```
void instanceofCheck(Base);
  0: aload_0
  1: instanceof Derived
  4: istore_1
  5: return
```
(d) Instanceof folding

Figure 6: Examples that show how OurInliner computes predicates for all the analyses that it currently supports.

argument of `nullCheck()` is the receiver of a call to `java.lang.Object.getClass()`. Upon entering `nullCheck()`, OurInliner places that value on the abstract local variable array. OurInliner models `aload_0` by loading the reference in the abstract local variable array at position 0 to the abstract operand stack. If the value is `null`, then this method always triggers an exception. If it is never `null`, then this call never triggers an exception. To model this behaviour, OurInliner generates the following inlining summary:

$$f_0(\hat{a}_0) = \hat{a}_0 \sqsubseteq \langle \texttt{null}, \top, \top \rangle$$
$$f_1(\hat{a}_0) = \hat{a}_0 \sqsubseteq \langle \texttt{notnull}, \top, \top \rangle$$

6.2.3 `Checkcast` *Folding.* The semantics of `checkcast` state that the object reference at the top of the stack must be of a type specified in the constant pool. If the reference at the top of the stack is `null`, then an exception is thrown. To model this behaviour for the example in Figure 6c, OurInliner generates the following inlining summary:

$$f_0(\hat{a}_0) = \hat{a}_0 \sqsubseteq \langle \textit{not null}, \top, \texttt{Derived} \rangle$$
$$f_1(\hat{a}_0) = \hat{a}_0 \sqsubseteq \langle \textit{null}, \top, \top \rangle$$

where f_0 corresponds to the case when the `checkcast` instruction can be folded because OurInliner has statically determined that the cast to `Derived` succeeds. The predicate f_1 corresponds to the case when the `checkcast` instruction cannot be folded. This case occurs when OurInliner determines that an exception will be thrown because the operand is always `null`.

6.2.4 `Instanceof` *Folding.* The semantics of `checkcast` and `instanceof` are similar. The main difference is that `instanceof` pushes a boolean value to the stack, while `checkcast` pushes a reference. To model this behaviour for the example in Figure 6d, OurInliner generates the following inlining summary:

$$f_0(\hat{a}_0) = \hat{a}_0 \sqsubseteq \langle \textit{not null}, \top, \texttt{Derived} \rangle$$
$$f_1(\hat{a}_0) = \hat{a}_0 \sqsubseteq \langle \textit{null}, \top, \top \rangle$$

where f_0 corresponds to the case when the `instanceof` instruction can be folded because OurInliner has statically determined that the return value is true. The predicate f_1 corresponds to the case when the `instanceof` instruction cannot be folded because OurInliner has statically determined that the return value is false.

6.3 Bringing It All Together

To formulate function inlining as a nested-knapsack problem, OurInliner computes a single positive integer to represent the abstract notion of benefit and a single positive integer to represent the abstract notion of cost. Similar to the majority of inliners [4, 5, 14, 22–24], OurInliner uses method size to abstract the cost of compilation without modifying the size of functions as a proxy for *inlineability*. Instead, OurInliner maintains the cost of inlining a method proportional to their sizes to prevent code growth caused by inlining large methods. To determine the abstract notion of benefit, OurInliner combines the direct and indirect benefits of inlining:

$$\textit{benefit} = \textit{root call ratio} \times (1 + \textit{indirect benefit})$$

This equation balances between the two components of the abstract notion of benefit. The root call ratio represents how likely the function executes once the control flow enters the compilation unit. The indirect benefits are an estimation for the benefits brought by optimizations that may be applied to the caller function after inlining. Therefore, the total benefit of inlining takes into consideration the frequency of execution of optimized code. The 1 is added to the expression to avoid multiplying by 0 in case OurInliner does not find any indirect benefits.

7 EVALUATION

To evaluate OurInliner, we have implemented a prototype of it as an experimental fork of the Eclipse OpenJ9 compiler infrastructure. We have also contributed our implementation to the open-source repository of Eclipse OpenJ9 [2]. Our evaluation compares that implementation with BaseInliner, the state-of-the-art JIT inliner in Eclipse OpenJ9, by attempting to answer the following research questions:

- **RQ1**: How does OurInliner affect compilation time?
- **RQ2**: How does OurInliner affect the generated code size?
- **RQ3**: How does OurInliner affect the program running time?

7.1 Machine Setup

We ran all experiments on an x86_64 machine with four 2.4 GHz AMD EPYC 7351 16-Core processors with a total RAM of 472 GB. The machine runs Ubuntu 18.04.4, Eclipse OpenJ9 VM version 0.23.0 (SHA: ee576d818), Eclipse OMR (SHA: 9f5372509), and a Java Class Library that is compatible with Java 1.8 (SHA: 03cb3a3 based on jdk8u232-b09). To avoid the effects of stop-the-world garbage collection events from dominating the execution of the benchmarks, we set the JVM heap to 1 GB. This setup is larger than the default, allowing the generational garbage collector enough space to operate without having to run a stop-the-world compact. Thus, we reduce the effect of completely pausing the running program to execute a garbage collection task. To avoid potential non-determinism that Non-Uniform Memory Access (NUMA) may introduce, we pinned the running process to use active cores from a single NUMA node.

7.2 Benchmark Setup

Our evaluation uses 10 DaCapo 9.12 Bach benchmarks [6] that run on JDK 8 [12]. To reach a stable state, each benchmark executes for several iterations until the JVM has been warmed up. The number of warmup iterations for each benchmark is based on the JIT compiler

Table 2: Warmup iterations for each DaCapo benchmark.

Benchmark	# Iterations	Benchmark	# Iterations
avrora	60	jython	100
eclipse	10	luindex	600
fop	1,000	pmd	100
lusearch	100	sunflow	100
h2	100	xalan	400

activity: the JVM has finished the warmup stage when it runs at least 3 iterations of the benchmark without any JIT compilation requests, or when it has not issued new compilation requests in the last 30 seconds of execution. Table 2 shows the number of warmup iterations for each benchmark.

After a benchmark reaches the end of the warmup stage, it runs one more time to record the program running time. To ensure that compiling a benchmark does not affect the measurements of other benchmarks, each benchmark runs in an isolated environment. To account for variations in execution time that may be introduced by background processes, each measurement is executed 10 times. For each benchmark, we report the arithmetic mean over those runs and the standard deviation based on modelling the recorded points for each run as a gaussian distribution. A single execution batch consists of running each benchmark once in a given order. To avoid bias in measurements due to variables not under control, we randomized the order of executing the benchmarks in each batch.

7.3 Compilation Time (RQ1)

The compilation time reported is the sum of the compilation time spent by each compilation thread in Eclipse OpenJ9. Figure 7 shows the average and the standard deviation for the total compilation time of each benchmark program, normalized to the average total compilation time that BaseInliner reports.

Across all analyzed programs, OurInliner is 14% (geometric mean) faster than BaseInliner. In particular, OurInliner compiles eclipse, fop, jython, luindex, lusearch, pmd, and xalan faster than BaseInliner, while it compiles h2 almost as fast as BaseInliner. The only exceptions are avrora and sunflow, where OurInliner spends more time compiling the benchmarks compared to BaseInliner (approximately 16% and 14%, respectively). These results show that, for most programs in our benchmark, OurInliner does not induce significant overhead to JIT compilation in comparison to traditional techniques, suggesting that OurInliner is viable within a JIT compiler.

7.4 Generated Code Size (RQ2)

Compilation time is related to the generated code size because most compiler optimizations have super-linear time complexity on the code size [5]. Therefore, in addition to evaluating the compilation-time overhead of OurInliner, we also evaluate the amount of compiled code that it generates. The code size for each compiled method body is the difference between the start and the end addresses of the method obtained through the verbose option in Eclipse OpenJ9. The total generated code size is the sum the generated code sizes for all the compiled method bodies. Figure 7 shows the average total generated code size of each benchmark program with its respective standard deviation across different runs, normalized to the average total generated code size that BaseInliner reports.

Across all analyzed programs, OurInliner generates 21% (geometric mean) less code compared to BaseInliner. In particular, OurInliner generates less code for all programs except avrora (approximately 2% more generated code). This large difference is due to BaseInliner conflating the notions of method size with method frequency information. This design favours inlining hot methods with large sizes, while consuming less of the inlining budget than considering only the original method size. In contrast, OurInliner does not alter the notion of method code size, but rather selects inlining candidates based on the combination of call-frequency information and potential post-inlining optimizations.

7.5 Program Running Time (RQ3)

The program running time is reported by the DaCapo benchmarking infrastructure for each iteration of the benchmark. We measure this time after each benchmark program has reached the warmup stage. Figure 7 shows the average and the standard deviation for the running time of each benchmark, normalized to the average program running time when using BaseInliner.

Across all analyzed programs, OurInliner causes only a 5% (geometric mean) increase in the program running time compared to using BaseInliner. This increase is dominated by the results for fop, h2, and jython, where OurInliner incurs approximately 20%, 11%, and 21% increase in program running time, respectively. For all other benchmarks, the program running time when using OurInliner is roughly similar to the case when using BaseInliner. To understand why OurInliner induces an overhead in the program running time for fop, h2, and jython, we conducted a performance analysis for these benchmark runs using the Linux `perf` command.

For fop and jython, the difference is due to an engineering decision about inlining in BaseInliner that happens outside of the inlining cost-benefit calculation. A module breakdown from the `perf` data for fop shows that OurInliner runs `j9vm29.so` for 8.5% of the time, while BaseInliner runs it for only 2% of its time. For jython, a module breakdown shows that the time that OurInliner spends running `j9vm29.so` is double that of BaseInliner. This time difference means that OurInliner spends more time interpreting code than BaseInliner. Upon analyzing the symbol breakdown from the `perf` data for these runs, we see that OurInliner shows `VM_BytecodeInterpreterCompressed::run` and `java.util.concurrent.ConcurrentHashMap.put()` at the top of its profile for fop and jython, respectively. These methods are at the bottom of the profile for BaseInliner. Further investigation shows that the main reason for this behaviour is that BaseInliner handles calls to `sun.misc.Unsafe.get()` and `sun.misc.Unsafe.put()` in a different way than other calls. In particular, BaseInliner always inlines these calls by marking them with a special flag that leads the code generator to recognize the methods and generate high-speed code for them, regardless of the decisions of the underlying inlining algorithm. On the other hand, OurInliner does not have any special handling for any method, and solely depends on the computed benefit of inlining based on its algorithm.

For h2, the difference is due to BaseInliner conflating the notion of method size to enable inlining larger methods than specified by

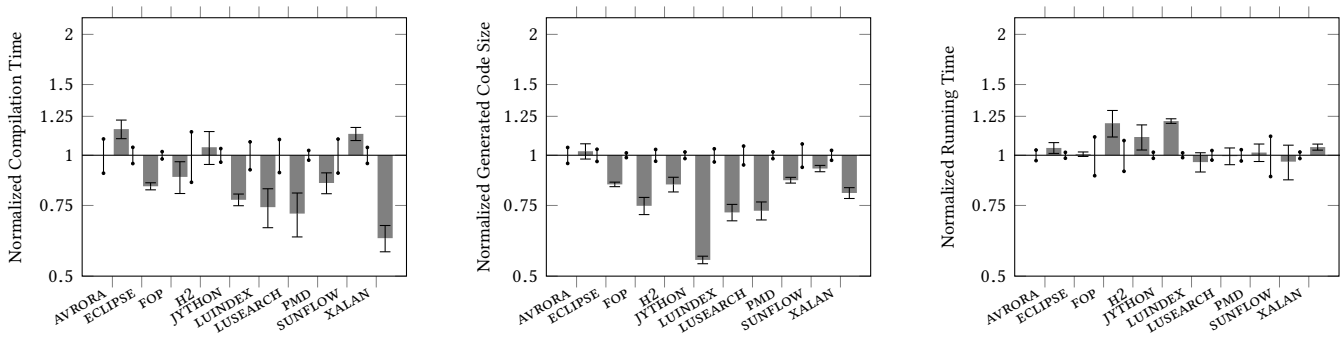


Figure 7: The compilation time, generated code size, and program running time for each of the analyzed DaCapo benchmarks when compiled by OurInliner, in logarithmic scale, normalized to that of BaseInliner. The error bars with (-) endpoints are for OurInliner, while those with (·) endpoints are for BaseInliner. Values smaller than 1 are better.

the input configuration. Additionally, BaseInliner employs a few heuristics that avoid inlining warm methods into other warm methods as well as inlining scorching methods into methods at lower optimization levels. A module and symbol breakdown from the `perf` data shows that OurInliner spends more time in the kernel, suggesting that the code generated by OurInliner has not enabled the underlying JIT compiler to perform its lock elimination optimization. Further analysis of the performance profile reveals that this is because OurInliner failed to inline a hot method because it exceeds the maximum inlining budget, causing the JIT not to reap the post-inlining benefits of inlining that method. This example indicates that further development effort in the modular design of OurInliner—to estimate the benefit of lock elimination in this case—would improve its runtime performance.

7.6 Discussion and Threats to Validity

7.6.1 Applicability of OurInliner. The goal of an inlining strategy is to improve the performance of the analyzed program in one of three aspects: compilation time, generated code size, or run-time performance. Our empirical evaluation has shown that, compared to BaseInliner, OurInliner achieves that goal for the first two aspects, but not necessarily for the third one. That is despite the limited engineering resources that we could put towards building OurInliner compared to the detailed fine tuning by domain experts that BaseInliner has seen over the years.

The current implementation of OurInliner supports a few simple static analyses that track constants and `null` values throughout the analyzed program, without resorting to any heuristics that BaseInliner has baked into its checks. We believe that future extensions of OurInliner (e.g., adding static analyses that model the heap) will only improve its performance to be on par with the industry-grade BaseInliner. Nevertheless, we still believe that the current implementation of OurInliner has its potential in some usage scenarios such as running in a cloud environment. In this environment, startup time and footprint are likely to be more important, for which OurInliner is already better than BaseInliner. Additionally, the inlining summaries that OurInliner computes allow compiler engineers to better understand why a certain inlining decision was taken by the JIT compiler, as opposed to the heuristics employed in other techniques. Compiler engineers would then be better equipped to debug, maintain, and improve their codebase.

7.6.2 Generality of Results. Since our primary focus is providing a new technique for Java JIT compilation, we chose to evaluate OurInliner compared to BaseInliner using the standard DaCapo benchmarks [6]. One might argue though that compilers typically operate on open programs (i.e., inputs are unknown) while these benchmarks are either closed programs (i.e., do not take an input) or have known inputs (e.g., lusearch and luindex). However, our choice facilitates a fair, reproducible comparison between OurInliner and BaseInliner where both operate on the same data every time to produce a deterministic behaviour. Since the algorithms within OurInliner do not depend on program inputs, we do not consider operating on a standard benchmark to be a limitation of our evaluation methodology.

Given that we have developed OurInliner only for Java, we cannot generalize our results to other languages that also compile to the JVM bytecode such as Scala, Clojure, and Groovy. The primary reason is that each of these languages implement the translation of the source language to JVM bytecode instructions in various ways that are quite different compared to Java [1]. For example, Groovy, Clojure, Python, and Ruby employ heavy use of reflection, `invokedynamic`, and run-time code generation. On the other hand, Scala, OCaml, and Scheme employ techniques that are similar to Java when compiling their source language to JVM bytecode instructions. We consider experimenting with those languages as a potential future direction for OurInliner that is orthogonal to the contributions that this paper presents.

8 RELATED WORK

There has been a large body of work on JIT compiler optimizations. This discussion is limited to related work about the main components of the inlining strategy.

Discriminants. Early work on inlining used call-site heuristics to select inlining candidates. In particular, Scheifler [21] use profile information at call sites to determine which functions to inline. Hazelwood and Grove [14] use the method size as a way to determine which methods to inline. Dean and Chambers [11] inline methods whose type group information indicates that an optimization may take place post inlining. Shankar et al. [23] proposes an inliner that assigns positive values to methods that, once inlined, will lead to decrease in object churn. Sewe et al. [22] favours methods that lead to further inlining without guards. Unlike discriminants used in that prior work, OurInliner presents a more general approach that

models call-site invariants, and their relationship with the method body, to obtain a more accurate model of potential post-inlining optimizations.

Prokopec et al. [19] have recently presented an approach that alternates between inlining and optimizations by incrementally exploring the program's call graph. This exploration is similar to exploring the IDT in OurInliner. However, our approach does not depend on fine-tuning heuristics through inlining trials which is a hard, time-consuming task that requires plenty of domain expertise.

Algorithm. Prior work has traditionally modelled inlining as a variant of the knapsack problem [3, 24]. Arnold et al. [4] and Shankar et al. [23] presents a greedy knapsack algorithm with different weights and values of inlining candidates. Other possibility for solving the knapsack problem includes dynamic programming, however, the dynamic-programming solution to the knapsack problem is not often used for inlining. Chang et al. [8] proves that inlining is a more complex problem than the knapsack problem. Therefore, no existing algorithm produces an optimal inlining plan. One of the novelties in the OurInliner approach is tomodel inlining as a nested knapsack problem [10], which allows it to model the mutual dependency between finding new inlining candidates and processing the newly found call sites in them.

Search Space. Most inlining strategies do not consistently define a notion of search space. Instead, they rely on previous inlining decisions to determine which call sites should be analyzed [3, 8, 14, 21, 24]. This inconsistency is due to the limitations of the greedy knapsack variants that inliners have traditionally used, which may lead to missing potential post-inlining optimizations. To the best of our knowledge, OurInliner is the first strategy that formulates a well-defined search space through the construction of the IDT.

9 CONCLUSION

Traditional JIT compilers rely on hard-coded heuristics and profile information to select inlining candidates. However, those heuristics are hard to maintain and require several years of domain expertise to fine tune. To overcome these limitations, we have presented OurInliner, a framework for Java JIT inlining that provides a systematic way of modelling the direct and indirect benefits of inlining. To achieve that, OurInliner employs abstract interpretation to generate inlining method summaries that it then applies to the estimated runtime values at call sites. To showcase the viability of OurInliner, we implemented a prototype for Eclipse OpenJ9 that supports a small set of static analyses. Compared to the state-of-the-art JIT inliner in Eclipse OpenJ9 and across the Java DaCapo benchmarking suite, OurInliner has 14% less compilation time and generates 21% less code. The current implementation, with significant less development effort than BaseInliner, incurs only 5% increase in program running time.

REFERENCES

[1] Karim Ali, Xioani Lai, Zhaoyi Luo, Ondřej Lhoták, Julian Dolby, and Frank Tip. 2019. A Study of Call Graph Construction for JVM-Hosted Languages. *IEEE Transactions on Software Engineering (TSE)* (nov 2019), (accepted to appear).

[2] Anonymous Authors. 2020. Eclipse OpenJ9. URL not shown to comply with the double-blind review process..

[3] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. 2005. A Survey of Adaptive Optimization in Virtual Machines. *Proc. IEEE* 93, 2 (2005), 449–466. https://doi.org/10.1109/JPROC.2004.840305

[4] Matthew Arnold, Stephen J. Fink, Vivek Sarkar, and Peter F. Sweeney. 2000. A Comparative Study of Static and Profile-Based Heuristics for Inlining. In *ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo)*. 52–64. https://doi.org/10.1145/351397.351416

[5] Paul Berube. 2012. *Methodologies for Many-input Feedback-directed Optimization.* Ph.D. Dissertation. Edmonton, Alta., Canada. Advisor(s) Amaral, Jose Nelson. https://doi.org/10.7939/R3DW8K AAINR89287.

[6] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. 169–190. https://doi.org/10.1145/1167473.1167488

[7] John Cavazos and Michael F. P. O'Boyle. 2005. Automatic Tuning of Inlining Heuristics. In *ACM/IEEE Conference on High Performance Networking and Computing*. 14. https://doi.org/10.1109/SC.2005.14

[8] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen-mei W. Hwu. 1992. Profile-guided Automatic Inline Expansion for C Programs. *Journal on Software: Practice and Experience* 22, 5 (1992), 349–369. https://doi.org/10.1002/spe.4380220502

[9] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*. 238–252.

[10] Andrew J Craik, Rachel E Craik, and Patrick R Doyle. 2017. Expanding Inline Function Calls in Nested Inlining Scenarios. US Patent App. 15/245,241.

[11] Jeffrey Dean and Craig Chambers. 1994. Towards Better Inlining Decisions Using Inlining Trials. In *ACM Conference on LISP and Functional Programming*. 273–282. https://doi.org/10.1145/182409.182489

[12] Eclipse OpenJ9. 2019. DaCapo Exceptions. https://github.com/eclipse/openj9/issues/4859.

[13] Eclipse OpenJ9. 2021. Eclipse OpenJ9. https://www.eclipse.org/openj9/.

[14] Kim M. Hazelwood and David Grove. 2003. Adaptive Online Context-Sensitive Inlining. In *International Symposium on Code Generation and Optimization (CGO)*. 253–264. https://doi.org/10.1109/CGO.2003.1191550

[15] Owen Kaser and C. R. Ramakrishnan. 1998. Evaluating Inlining Techniques. *Computer Languages* 24, 2 (1998), 55–72. https://doi.org/10.1016/S0096-0551(98)00003-4

[16] Sorin Lerner, David Grove, and Craig Chambers. 2002. Composing Dataflow Analyses and Transformations. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. 270–282. https://doi.org/10.1145/503272.503298

[17] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2014. The Java Virtual Machine Specification-Java SE 8 Edition.

[18] Steven S. Muchnick. 1997. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[19] Aleksandar Prokopec, Gilles Duboscq, David Leopoldseder, and Thomas Würthinger. 2019. An Optimization-Driven Incremental Inline Substitution Algorithm for Just-in-Time Compilers. In *International Symposium on Code Generation and Optimization (CGO)*. 164–179.

[20] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. 49–61. https://doi.org/10.1145/199448.199462

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

[22] Andreas Sewe, Jannik Jochem, and Mira Mezini. 2011. Next in Line, Please!: Exploiting The Indirect Benefits of Inlining by Accurately Predicting Further Inlining. In *Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH)*. 317–328. https://doi.org/10.1145/2095050.2095102

[23] Ajeet Shankar, Matthew Arnold, and Rastislav Bodík. 2008. Jolt: Lightweight Dynamic Analysis and Removal of Object Churn. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. 127–142. https://doi.org/10.1145/1449764.1449775

[24] Edwin Steiner, Andreas Krall, and Christian Thalinger. 2007. Adaptive Inlining and On-Stack Replacement in the CACAO Virtual Machine. In *International Symposium on Principles and Practice of Programming in Java (PPPJ) (ACM International Conference Proceeding Series)*, Vol. 272. 221–226. https://doi.org/10.1145/1294325.1294356

[25] Peng Zhao and José Nelson Amaral. 2003. To Inline or Not to Inline? Enhanced Inlining Decisions. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC) (Lecture Notes in Computer Science)*, Vol. 2958. Springer, 405–419. https://doi.org/10.1007/978-3-540-24644-2_26