Guiding Inlining Decisions Using Post-Inlining Transformations

by

Erick Eduardo Ochoa

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Inlining strategies in just-in-time (JIT) compilers have relied on a mixture of heuristics and frequency information to discriminate between inlining candidates. Even though nested inlining is the norm in JIT compilers, modeling inlining as a variation of the greedy knapsack algorithm provides sub-optimal solutions. Recent advancements by Craik et al. [16] allow for solutions for the nested inlining problem, however, they require grading inlining candidates based on an abstract notion of "benefit". In this thesis, we define this abstract notion of "benefit" through the use of static analysis and frequency information. The choice of using static analysis instead of heuristics has consequences on the guarantees that an inlining strategy provides and on the compilation time incurred by the compiler. We show that our proposed static analysis is comparable to heuristics in terms of compilation time, memory resources used during compilation, and impact on run time. While our proposed inlining strategy increased run time by 4% compared to the default inlining strategies found in the OpenJ9 Java Virtual Machine (JVM), our inlining strategy allows compiler engineers to fine tune the abstract notion of "benefit" and provides human readable reports that show why inlining decisions were taken. The number of proposed inlining plans that differ between the heuristics and the static analyses is small, however, it provides a lower bound for how more powerful static analyses may improve inlining in the future.

Preface

We intend to publish the results of this thesis after future work and known improvements have been made. The contents of Chapters 4, 5, and 6 fall under a patent filed with the U.S Patent Office as A. Craik, **E. Ochoa**, J. N. Amaral, K. Ali "Assessment of the benefit of post-inlining program transformation in inlining decisions", Patent Reference "P201803683US01", filed on April 30, 2019. I was responsible for designing, building, and experimentally evaluating the framework within the OpenJ9 JVM. A. Craik contributed to many technical discussions and OpenJ9 specific advice. K. Ali was the supervisory author and contributed to guiding the research direction of the project. J.N. Amaral was the co-supervisor and also contributed to guiding the research direction of the project. Many thanks to A. Craik, J.N. Amaral and K. Ali for proofreading and help editing this thesis.

Another research result, which is not part of this thesis, but was completed during my M.Sc program at the University of Alberta is a research project published as T. Lloyd, A. Chikin, **E. Ochoa**, K. Ali, J.N. Amaral, "A Case for Better Integration of Host and Target Compilation When Using OpenCL for FPGAs", *Proceedings of Fourth International Workshop on FPGAs for Software Programmers*, in October 2017. This work introduced a series of interconnected compiler transformations aimed at improving performance of Field-Programmable Gate Array (FPGA) programs generated through high-level synthesis of Open Compute Language (OpenCL). My roles on the project included setting up and maintaining required infrastructure, implementing the host analysis and transformation optimization passes, and the evaluation of the project.

Another research result, which is not part of this thesis, also completed

during my M.Sc program is a research project published as J.N. Amaral, E. Borin, D. Ashley, C. Benedicto, E. Colp, J.H.S. Hoffmam, M. Karpoff, **E. Ochoa**, M. Redshaw, and R.E. Rodrigues, "The Alberta Workloads for the SPEC CPU 2017 Benchmark Suite", *2018 IEEE International Symposium on Performance Analysis of Systems and Software* in April 2018. This work introduces alternative workloads for the SPEC benchmarking suite aimed at correcting overfitting in profile driven optimization evaluations. My roles on the project included creating new workloads for the `omnetpp` benchmark, mantaining infrastructure related to the evaluation, and running evaluation scripts for multiple benchmarks with the new workloads.

To my family.

Acknowledgements

I would like to thank my supervisors, Dr. Karim Ali and Dr. J. Nelson Amaral, for their patience and support throughout these three years. Their guidance and support cannot be understated.

I would like to thank Andrew Craik for lending his expertise uncountable number of times and answering so many of my questions promptly.

I would also like to thank to my friends and co-workers at the University of Alberta (listed in alphabetical order): Zaheen Ahmad, Quinn Barber, Artem Chikin, Shrimanti Ghosh, Adriana Hernandez, J. Fernando Hernandez, Ashley Herman, Martin I. Oliveira, Ifaz Kabir, Marcus Karpoff, Braedy Kuzma, Gustavo Leite, Bernard Llanos, Taylor Lloyd, Fernando Lopez de la Mora, Mehran Mahdi, John Wood, and others that I've failed to mention. Thanks for the many technical insight and discussions we had. But most importantly, thank you for making these years fun.

I gratefully acknowledge the funding support of the IBM Center for Advance Studies (CAS) Canada, the Government of Alberta, and the University of Alberta.

Contents

List of Tables

List of Figures

List of Acronyms

AOT Ahead Of Time.

CG Call Graph.

IDT Inlining Dependency Tree.

JIT just-in-time.

JVM Java Virtual Machine.

LIFO last-in-first-out.

NUMA Non-Uniform Memory Access.

OS operating system.

VM Virtual Machine.

Chapter 1

Introduction

Function inlining is a program transformation that replaces a call site with the body of the function being called. It is considered an important transformation for two main reasons: (1) it allows the compiler to eliminate the overhead of invocation and frame allocation costs, and (2) it allows the compiler to further optimize the inlined function into its calling context. These benefits are known as the *direct benefits* and the *indirect benefits* of inlining, respectively. Inlining is key in reducing run time because of these two benefits and in some cases it may also help reduce program's size. However, function inlining also has its drawbacks: (a) applying function inlining indiscriminately may lead to slower executions [44], (b) applying it to all invocations will never terminate for recursive functions [39], and (c) inlining usually leads to an increase in compilation time, and costs in program storage and transmission [10]. Therefore, function inlining must be applied selectively.

Many strategies have been designed to apply inlining selectively. Each of these strategies has its tradeoffs in the following areas: (a) compilation time, (b) quality of generated code, (c) used resources, (d) maintainability, and (e) extensibility. Contemporary JIT inlining strategies focus on estimating the direct benefits of inlining using profile information and estimating the indirect benefits of inlining by using heuristics [5], [25], [44]. These heuristics include number of arguments, size of methods, inlining based on constantness of arguments, discriminating against inlining polymorphic methods, and not considering inlining beyond a certain stack depth. While heuristics may

provide a fast decision procedure to determine whether or not to inline a function, heuristics may be difficult to generalize for different workloads [44]. Heuristics also lead to suboptimal decision making [25]. Furthermore, source-level heuristics "do not consider the effect of optimizations applied to the body of the called routine after inlining" [18].

Researchers have been pushing for analysis-driven inlining that provides guarantees on the inlining process. For example, Hazelwood et al. [25] propose an analysis to conditionally eliminate the weight given by a heuristic to an inlining candidate. The weight assigned to an inlining candidate is restored (as if the heuristic did not exist) if the inlining candidate has not taken advantage of the static information provided by the heuristic [25]. Also, Shankar et al. [41] propose an analysis to assign weights to inlining candidates depending on an analysis to decrease the amount of objects allocated on the heap. However, these analyses model only the semantics of interests (i.e., whether an object will escape the stack, or whether devirtualization is likely to happen).

Dean et al. propose an inlining strategy that tentatively inlines a procedure and matches the static information found in the caller with the optimizations that took place [18]. Their analysis constrains optimizations to type of arguments. The optimizations being considered are determining the targets of dynamic dispatches, and constant propagation. However, their analysis performs inlining before the analysis is capable of determining which optimizations will take place. They resolve this issue by creating a database of optimizations which is constrained by the type of the arguments. Inlining before the database is populated might result in suboptimal decisions.

This work generalizes analysis to calculate which compile time optimizations will take place (like inlining trials). However, unlike inlining trials, the calculation is obtained without performing an inlining trial. This thesis investigates

1. the compilation cost and the impact of running static analyses in a JIT context

2. can static analysis be used to guide inlining decisions based on the indirect benefits of inlining

The results of an experimental evaluation show that a fast static analysis can be applied in the JIT context. The run time and memory usage is comparable with state of the art inlining strategies. While it is possible to take advantage of the static information provided in the code, the applicability of the analysis is limited by its precision. We show that in industry benchmarks there are few examples were our proposed inliner can take advantage of static information to make inlining decisions. However, our inliner has other properties such as extensibility and explainability which is of interest to compiler developers.

The contributions of this thesis include:

1. A fast framework for performing static analysis on Java bytecode.

2. Guarantees on the optimizations unlocked after inlining.

3. A ongoing open source contribution to the OpenJ9 project.

This thesis is organized as follows: Chapter 2 presents background information on the JVM, JIT compilation, abstract interpretation, the knapsack problem and their relation to inlining. Chapter 3 discusses related work and provides a model to categorize different inlining strategies according to several properties. Chapter 4 gives an overview of our proposed selective inlining strategy, outlining how abstract interpretation and constraints take a part in determining which optimizations will take place after inlining. Chapter 5 gives in detail the semantics given to the java bytecode in order to estimate arguments at call sites. Chapter 6 outlines the optimizations which are determined at analysis time and how they are encoded into method summaries. Section 6.6 combines the processes described in Chapter 5 and Chapter 6 in order to give a single numeric value to inlining candidates. Chapter 7 describes the evaluation process and the limits of the proposed inlining strategy. Finally, the conclusion and future work is found on Chapter 8.

Chapter 2

Background

This Chapter, discusses several properties of the JVM that are important for understanding this thesis. Furthermore, it introduces the OpenJ9 JVM, which is the platform where a prototype for the analysis is implemented. The profiling frameworks and `VPConstraint` framework available in OpenJ9 are outlined as they both contribute to the scoring of inlining candidates. Data-flow problems and abstract interpretation is related back to the `VPConstraint` framework. Differences between inlining static functions and virtual functions and differences in solutions to the knapsack problem are discussed.

2.1 The Java Virtual Machine

A JVM is is a Virtual Machine (VM) that enables a computer to run Java programs as well as programs written in other languages that are compiled to Java bytecode [46]. Different runtime services such as garbage collection and native interfaces, are part of this execution environment [17]. While its interface is thoroughly defined, its implementation is loosely defined to allow different implementations. As a result of this, there are many different implementations of JVMs including: HotSpot [27], OpenJ9 [22], Azul Zing JVM [8], Jikes RVM [3], Kaffe [34], Maxine [9].

2.1.1 Just in Time Compilation

The Java Virtual Machine Specification does not specify the execution mode of the bytecode [31]. Because the bytecode is architecture independent, initially

the JVM used interpreters to execute applications. However, even optimized Java interpreters perform poorly when compared to compiled code [17]. As a result, a majority of JVMs now use JIT compilation to reduce the overhead introduced by interpretation. JIT compilation (or dynamic compilation) is an optimization technique that compiles architecture independent bytecodes into native code [7].

A JVM invokes the method-based JIT compiler, like the one found in OpenJ9, at the method level. Initially the JVM interprets all methods, but when a given method is executed multiple times, the JVM triggers a JIT compilation of the method. In some JVMs multiple re-compilations of a given method, at increasing optimization levels, may be requested as the frequency of execution of the method increases. At each level different code transformations are attempted.

2.1.2 The Java Bytecode

JVMs execute architecture-independent bytecodes. The stack-based bytecode used by the JVMs is usually obtained through compiling Java source into bytecodes by a Java compiler. However, other languages and compilers also target the JVM. The semantics of the bytecodes is described in the Java Virtual Machine Specification [31].

The Java SE Specification [31] defines the following primitive data types supported by the Java Virtual Machine:

- `byte`, whose value are 8-bit signed two's-complement integers
- `short`, whose values are 16-bit signed two's-complement integers
- `int`, whose values are 32-bit signed two's-complement integers
- `long`, whose values are 64-bit signed two's-complement integers
- `char` whose values are 16-bit unsigned integers
- `float` whose values are of the float value set or, where supported, the float-extended-exponent value set

- `double` whose values are elements of the double value set or, where supported, the double-extended-exponent value set
- `boolean` which have limited support in the JVM, and are converted to `int` s
- `returnAddress` which is used by the JVM's *jsr*, *ret*, and *jsr_w* instructions.
- `references`, which can point to arrays, classes or interfaces

2.1.3 Run Time Structures in the JVM

Instructions operate on several runtime data structures that are defined in the Java Virtual Machine Specification. Of particular importance for this work, we have the operand stack, the local variable array, and the stack frames.

Operand Stack

According to the Java Virtual Machine Specification:

> Each frame contains a last-in-first-out (LIFO) stack known as its *operand stack*.
>
> The operand stack is empty when the frame that contains it is created. The [JVM] supplies instructions to load constants or values from local variables or fields onto the operand stack. Other Java Virtual Machine instructions take operands from the operand stack, operate on them, and push the result back onto the operand stack. The operand stack is also used to prepare parameters to be passed to methods and to receive method results.

The JVM is designed with security in mind and as such certain properties are guaranteed to hold over the bytecode. Of particular importance is the fact that "stack depth is known at every branching point, and two execution paths merging at the same merge point must also have the same stack depth" [1]. This property is further outlined in the Java Virtual Machine Specification [31, Chapter 5].

Local Variable Array

> Each frame contains an array of variables known as its *local variables*. The index of the first local variable is zero.

Frames

> A *frame* is used to store data and partial results, as well as to perform dynamic linking, return values for methods, and dispatch exceptions.
>
> A new frame is created each time a method is invoked. [...] Each frame has its own array of local variables, its own operand stack, and a reference to the runtime constant pool of the class of the current method. The sizes of the local variable array and the operand stack are determined at compile-time and are supplied along with the code for the method associated with the frame.

2.2 Eclipse OpenJ9

In 2018 IBM open sourced the OpenJ9 JVM as the Eclipse OpenJ9 project [22], [26]. It includes an open sourced subset of IBM's Testarossa JIT [24]. Eclipse OpenJ9 uses selective dynamic asynchronous compilation. The VM selects methods to compile by inspecting method invocation counters and sampling the execution stack. The methods selected for compilation are then placed into the compilation queue. Multiple compilation threads query the compilation queue for work while the VM is asynchronously running the application.

Dynamic asynchronous compilation uses the profiling and the value propagation constraint frameworks. The next sections describe these frameworks in detail.

2.2.1 `VPConstraints`

Value propagation is a compiler transformation that propagates the definition of constant values to their uses. Value propagation can help alleviate register pressure by removing register use, replacing register use with immediate values,

```
1 a = 0
2 if (p) a = 10
3 else a = -10
4 return a
```

Figure 2.1: Example of value propagation in which only range propagation is possible.

and pruning entire unreachable program paths which simplifies subsequent analysis. In some cases, the analysis is not interested in just propagating the exact value but propagating a safe approximation to the value.

The `VPConstraints` library is a set of `C++` classes that are available in the OMR project [21]. These set of classes model potential types and values that can be expected at run-time.

For example, a value of type short is modeled by the `VPShortConstraint` class which is a class that holds two integers that correspond to the lower and upper bound that the value may hold.

For example, in Figure 2.1, the value returned can either be 10 or -10. The value depends on the condition `p`, which is not known at analysis time. Since `VPShortConstraint` work by modeling integer values as ranges, using the `VPConstraints` library would indicate that the return value could be any of the values in the range of $[-10, 10]$.

Some operations that are valid with `short` data types are also modeled . For example, the methods `add()` and `subtract()` take another `VPShort-Constraint`. In this case, if the value 5 is added to the previously calculated value, we would obtain the range $[-5, 15]$. Classes can be modeled according to constraints on the class hierarchy. The least precise estimate for a class is the Java primitive object `Object`. This corresponds to the $\top$ of the semi-lattice which models classes. The rest of the semi-lattice corresponds to the class hierarchy.

2.2.2 Profiler

Eclipse OpenJ9 has two different active profiling frameworks, the IProfiler, the JITProfiler [6], [23]. These two sources of profile information are used during inlining and compilation to determine which targets of virtual call sites are visited often, which leads to determining hot regions of code.

The IProfiler framework is the least expensive profiling framework available in OMR. It records branch biases in methods and the percentage of virtual call-site targets during interpretation of Java bytecodes. The branch biases are used to compute an estimate of basic block frequencies. Eclipse OpenJ9 uses this information during inlining to determine which call sites are frequently executed. The basic block frequencies are used to avoid inlining methods that are called from basic blocks that have not been visited .

The JITProfiler is the most expensive profiling framework available in OMR. The VM may decide to profile using the JITProfiler framework only if the method is executed frequently enough. The JITProfiler compiles an instrumented version of a method to record accurate basic block frequencies and virtual method receiver types. The instrumented version undergoes limited optimization and is interleaved with the corresponding uninstrumented version of the code. The execution of the instrumented version is interleaved with an optimized version of the code. A new low-overhead profiling framework is currently in development by the OpenJ9 team [15].

2.3 Data-flow Problems

Data-flow problems are a kind of program analysis task that asks "what kind of values a variable may hold at a particular point in the program" without running the program itself. Data-flow problems are generally undecidable and are closely related to the halting problem [30]. However, while data-flow problems generally cannot be solved precisely, a solution to them can be estimated safely and in finite time.

Different program optimization questions can be formulated as data-flow problems. For example, which variables are alive, reachable expressions, and

which statements are dead can all be formulated as data-flow problems. The implementation of these analyses would differ in what equations are used to model instructions, and what domain these equations operate on. For example, in variable liveness problems, the domain of the data-flow functions is $\mathcal{P}(v)$ (i.e., the power set of variables). In reachable definitions, the domain of the data-flow functions is $\mathcal{P}(v \times L)$ (i.e., the power set of the cross product of variables and labels in the program).

Reachable definitions and live variables would also differ on the equations used to model instructions. For example, reachable definitions and live variables model the assignment instruction. However, reachable definitions propagates the information of variable definition to succeeding program points, while variable liveness is interested in propagating variable assignments to preceding program points. As such, how these analyses model the assignment instruction's impact on the program's state would differ.

Solutions to data-flow problems typically model data-flow problems as a recursive set of equations. Each equation represents a program state transformation and takes as an input the program state at program point p and outputs the program state at program point $p+1$. These equations are typically called *flow functions*. These equations are restricted to operate on *lattices*, and should be *monotone*. Different solutions for data-flow problems exist and vary on their generality, complexity and precision [14], [30], [33], [35], [37], [38].

The set of recursive equations to be solved for Figure 2.2 can be found on Figure 2.3 In this example, we want to find out what definitions (a tuple of the form, (v, L) reaches the program points p. All nodes with multiple incoming edges must perform the join operation ($\sqcup$) on the values supplied by the incoming edges. The join operation find the greatest lower bound for two elements in the lattice. After that, all statements which are not assignment statements will use the identity flow function (where the output data-flow facts equal the incoming data-flow facts). Only the assignment statements will have a flow function of the form $f(X) = X \sqcap K \sqcup G$ where K and G are sets of data-flow facts determined by the statements. K and G are normally called in the literature the *Kill* and *Gen* sets because the K set will remove data-flow

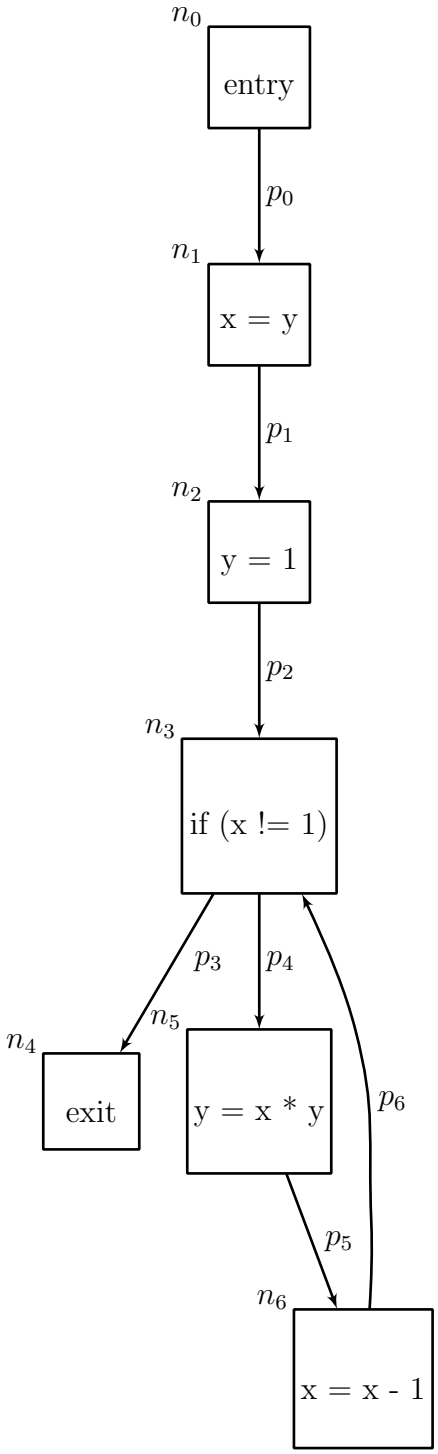


Figure 2.2: Factorial pseudocode

$$
\begin{aligned}
\emptyset &= p_0 \\
p_0 \sqcap K_x \sqcup \{(x, n_1)\} &\sqsubseteq p_1 \\
p_1 \sqcap K_y \sqcup \{(y, n_2)\} &\sqsubseteq p_2 \\
p_2 \sqcup p_6 &\sqsubseteq p_3 \\
p_2 \sqcup p_6 &\sqsubseteq p_4 \\
p_4 \sqcap K_y \sqcup \{(y, n_5)\} &\sqsubseteq p_5 \\
p_5 \sqcap K_x \sqcup \{(x, n_6)\} &\sqsubseteq p_6 \\
&\textit{where} \\
K_x &= \{(y, n_0), (y, n_1) ... (y, n_6)\} \\
K_y &= \{(x, n_0), (x, n_1) ... (x, n_6)\}
\end{aligned}
$$

Figure 2.3: Data flow equations to solve for finding reachable definitions.

facts from X and G will generate data-flow facts that will be in union with the result of $X \sqcap K$. The meet operation ($\sqcap$) on a lattice finds the least upper bound for two elements in the lattice.

$$P_{t+1} = \mathbb{S}(P_t)$$

Iterative solutions to data-flow problems can be seen as a function $\mathbb{S}$ that takes an approximate solution P_t for all program points p_i and returns a better approximate solution P_{t+1}. Iterative solutions stop when a fixed point (i.e., $P_{t+1} = P_t$ is found). Because the flow functions are monotonic and the analyses operate on a finite height lattices, a solution is guaranteed to exist [30].

2.3.1 Lattices

A lattice [19] is any set that has a binary relation, $\sqsupseteq$, between the elements and satisfies the following five properties:

1. the binary relation $\sqsupseteq$ is reflexive;

2. the binary relation $\sqsupseteq$ is transitive;

3. the binary relation $\sqsupseteq$ is anti-symmetric;

4. there is a unique least upper bound between any two elements in the set, $A \sqcup B$; and

5. there is a unique greatest lower bound between any two elements in the set, $A \sqcap B$.

Bounded lattices also have a $\top$ and a $\bot$ element. The $\top$ element is indempotent for the least upper bound operator. Similarly, the $\bot$ element is indempotent for the greatest lower bound operator. In data-flow analyses with bounded lattices, the $\top$ element is usually interpreted as the union of all possible values. Similarly, the $\bot$ element is usually interpreted as an undefined value.

2.3.2 Abstract Interpretation

Abstract interpretation is the theory of data-flow problems [14]. It formalizes the relationship between *concrete values* (i.e., those used in the program) and the *abstract values* (i.e., those used in the analyses). This formalization is known as the Galois connection. Abstract interpretation also specifies a framework for solving data-flow frameworks by modeling the *concrete semantics* of a program by approximating them using *abstract semantics*. The abstract semantics operate on abstract values and provide a safe estimate of the program's execution.

2.3.3 Interesting Run-Time Properties of Programs

Abstract interpretation can estimate interesting run-time properties of programs. Branch tests, the classes of references in the variable array, or the null-ness of arguments are run-time properties of interest. These run-time properties of programs are useful because they might lead to program optimization or aid the user in debugging. For example, if an analysis is capable of determining the value of branch tests, then that branch may be folded away. If a class of a reference is known, then it is possible to replace a dynamically-dispatched procedure calls with a direct call. These interesting run-time properties are all properties that can be obtained by inspecting the operand stack or the variable

array. Fortunately, abstract interpretation allows us to safely estimate these program properties.

2.4 Inline Substitution

Inline substitution or function inlining is the replacement of a function call with the body of the function being called. While this definition is usually sufficient for the general understanding of inlining, it omits a lot of details that need to be explained when introducing other definitions like inlining candidate and inlining decision.

2.4.1 Inlining Non-virtual Functions

Non-virtual functions are functions that are resolved statically (i.e., at compile time). The target of non-virtual function calls is unique and non-ambiguous. As a result, the previous definition for inline substitution is enough to explain how inlining of non-virtual functions work. A function call would be equivalent to an inlining candidate as for all non-virtual functions we can find the target of the function and inline it. An inlining decision is just a yes-or-no decision on whether the target of the non-virtual function replaces the call site.

2.4.2 Inlining Virtual Functions

Virtual functions are functions that are resolved dynamically (i.e., at run-time). Virtual function calls may have multiple potential targets for each call. As a result, virtual functions preclude inlining without safety checks.

Substituting the function call with any of the target functions is unsafe unless there is a mechanism to resolve the call dynamically. One way of doing safely resolving a virtual call is through method test [29]. A method test can be understood as an if-else statement that tests for a particular method. If the condition in the if-statement is true, then the path that is executed contains the body of the target method. If the condition is false, then the path that is executed contains the virtual function call to be resolved at run-time. We expand the definition of inlining to allow the inlining of virtual function calls.

```
R0 := <receiver object>
R1 := load(R0 + <offset-of-class-in-object>)
R2 := load(R1 + <offset-of-method-in-class>)
if (R2 == <address-of-inlined-method>) {
 <method inlining>
} else {
 call R2
}
```

Figure 2.4: Pseudocode for method test obtained from [29]

```
R0 := <receiver object>
R1 := load(R0 + <offset-of-class-in-object>)
R2 := load(R1 + <offset-of-method-in-class>)
if (R2 == <address-of-inlined-method-1>) {
 <method-1 inlining>
} else if (R2 == <address-of-inlined-method-2>) {
 <method-2 inlining>
} else {
 call R2
}
```

Figure 2.5: Pseudocode for method test with multiple virtual functions inlined.

As a result, an inlining candidate now becomes a call-site-callee-method pair. This generalization implies that there are multiple inlining candidates for a single call site. The inlining decision is still just a yes-or-no decision on which of the targets are inlined.

2.4.3 Inlining Multiple Virtual Functions

It is possible for the method test to allow for the inlining of multiple virtual functions at one particular call site. In this case, the method test is now not just a simple if-else statement but it is a chain of if-statements followed by an else-statement. We expand the definition of inlining to allow multiple target functions be inlined to call sites to virtual functions.

Reducing the overhead of the calling convention allows programs to finish execution in less amount of time.

2.5 The Knapsack Problem

The Knapsack Problem is a well known optimization problem where: (a) there exists a set of objects with a value and a weight, (b) another empty set (the knapsack) which is to be filled with the objects, and (c) a constraint on the objects' total weight in the knapsack. There exists multiple strategies on how to solve the knapsack problem. Understanding the solutions to the Knapsack Problem is important in the context of inlining because many inlining strategies model solutions to the Knapsack Problem. In this section, we will give a general introduction to the different solutions to the Knapsack Problem without relating it to inlining. In Chapter 3, these different solutions to the inlining problem will be referenced.

2.5.1 The Greedy Solution to the Knapsack Problem

A linear relaxation of the knapsack problem yields a greedy solution. Linear relaxation in this context implies that a fraction of the objects may be added to the knapsack instead of considering the item as a whole. The Greedy Solution to the Knapsack Problem is simple: sort the objects by the ratio of their value and their weight, then add objects in descending order and stop when the total weight is matched. Iterating over the sorted objects has a time complexity of $\mathrm{O}(n)$.

2.5.2 The Dynamic Programming Solution to the Knapsack Problem

The Greedy Solution to the Knapsack Problem does not give optimal solutions when linear relaxation is applied. The Dynamic Programming Solution explores the set of partial solutions to the Knapsack Problem. The Dynamic Programming Solution to the Knapsack Problem has an asymptote of time $\mathrm{O}(nw)$ in the time required for its execution, where w is the constraint on the weight the knapsack is allowed to carry.

2.5.3 Solving the Nested Knapsack Problem

Variations on the Knapsack Problem exist. Besides the linear-relaxation Knapsack Problem and the non-linear-relaxed Knapsack Problem, there is also a variation of the Knapsack Problem known as the Nested Knapsack Problem. The Nested Knapsack Problem is stated as follows: (a) there exists a set of objects with a value and a weight, (b) there exists a hierarchical relationship between the objects, (c) another empty set (the knapsack) which is to be filled by the objects, (d) only objects whose predecessors have been added to the knapsack may be added to the knapsack, and (e) a constraint on the objects' total weight on the knapsack.

The Dynamic Programming Solution for the Knapsack Problem presented in Section 2.5.2 is modified by Craik et al. as follows [16]: (a) the input to the algorithm is a list of functions to process in postorder over the hierarchical relation, (b) the list must also be considered in order of lowest to highest cumulative benefit, (c) backtracking over the partial optimal solutions, and (d) another backtracking step over a backwards traversal up a column of uniform cost to identify a previous less optimal solution to augment.

The Nested Knapsack Solution presented by Craik et al. has a time complexity of $\mathrm{O}(nwh)$, where h is the average depth of the hierarchy tree being considered [16].

2.5.4 Inlining Dependency Tree

Previous inlining algorithms iterated over an abstract program representation known as the Call Graph (CG). One of the break throughs of the inlining strategy proposed by Craik et al. [16] is that the iteration happens on a tree-like data structure as opposed to a graph-like data structure. Allowing the caller-callee relationship to be represented as a tree as opposed to a graph allows the dynamic algorithm to consider callsite-method pairs. In a graph, callsite-method pairs are represented as edges and a naive representation does not allow for context sensitivity. The tree-like data structure presented by Craik et al. [16] is called the IDT. The IDT is a data structure that models

```
1 A() {
2   B()
3   B()
4   C()
5   C()
6 }
7 B() {
8 }
9 C() {
10  B()
11 }
```

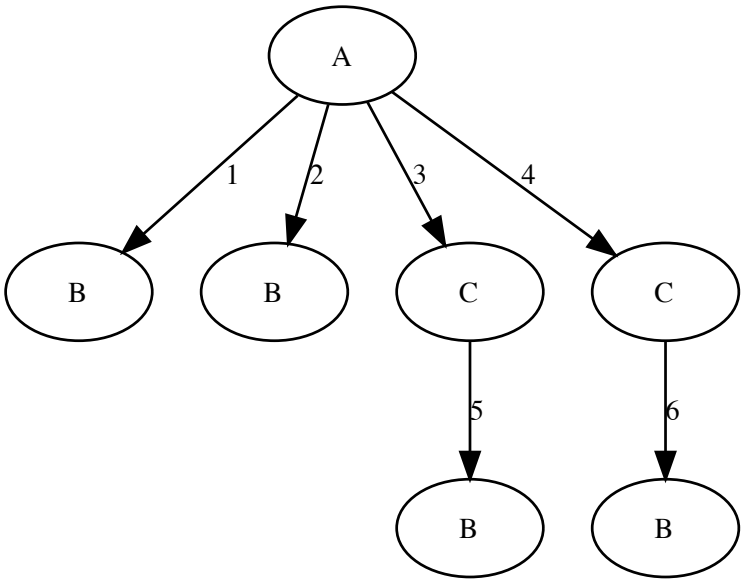
Figure 2.6: Pseudo-code to illustrate properties of an IDT

inlining decisions and, in the absence of partial inlining, corresponds directly to a call tree. Call sites are modeled as directed edges and methods as nodes in a tree. With one edge per call site, the IDT differentiates between call sites to the same callee by creating distinct nodes. The caller-callee relationship is maintained by the directionality of the edges with callers pointing to callees. Unlike a CG, nodes in the IDT only contain a single parent and do not loop. This tree-like structure preserves context sensitivity in the inlining decisions.

Let us take a look at example pseudocode in Figure 2.6 considering that method `A` defined in line 1 as the root of the IDT. First, we note that there are two distinct call sites for method `B`. One call site is at line 2 and another is at line 3. Let us enumerate the call sites to give them unique names. As such, the call site at line 2 is 1, and the call site at line 3 is 2. In Figure 2.7, we can see two edges starting from node `A` to two distinct nodes labeled `B`. There are two other nodes labeled `C` to denote the call-site method pair in lines 4 and 5. Node `C` also contains a call to method `B`.

As mentioned earlier, the IDT allows us to make context sensitive inlining decisions by referring to the call site indices. We can choose to inline the call at call site index 5 as opposed to inlining method `B` at all call sites, or even inlining method `B` when called by method `C`. The edges in Figure 2.7 are numbered only for the purposes of disambiguiation in this text. In the concrete implementation of the IDT, edges are not numbered but nodes in the IDT

Figure 2.7: IDT corresponding to Figure 2.6



contain the bytecode offset where the callsite was found. Storing the bytecode offset of the callsite allows one to differentiate between different callsites to the same method from the same caller.

2.6 Summary

We will be working on the Eclipse OpenJ9 JVM in order to determine which inlining candidates should be inlined. An analysis which relies on the concepts of abstract interpretation, the class `VPConstraints` and the profiling frameworks available in OpenJ9 has been developed. Unlike other inlining strategies (which are reduced to the knapsack problem), our solution leverages recent innovations in the solution to the nested knapsack problem. As such, we will be working on the IDT as opposed to other abstract program representations. This allows us to make context sensitive decisions on the inlining process.

Chapter 3

Related Work

Function inlining is an optimization that touches many different subsystems and concepts from compilers. In Section 3.1 we outline previous literature in inlining research. The research is categorized in what we believe is a model that explains the differences in inlining. Inlining research can be explained by looking into the concepts and different subsystems that inlining can influence or that are influenced by inlining. The properties of inlining that our model examines are:

1. *Problem formulation.* Inlining is often modeled as a knapsack problem, but the literature also presents alternative formulations.

2. *Algorithmic differences.* There are different algorithms that can solve the knapsack problem and they have trade-offs in run time, memory usage, and the constraints handled.

3. *Implementation differences.* Even within the same algorithms, there can be room for implementation-specific details that may result in different inlining decisions.

4. *Goal differences.* Normally inlining tries to minimize the run time of a program, however there may be different goals for inlining.

5. *Goal estimation differences.* There is only one way to measure the impact of an inlining decision, and that is running the program. As such, in order to discriminate between different inlining candidates, inlining strategies

estimate the impact of an inlining decision. Different inlining strategies may have the same goal but a different way to estimate the impact of inlining in achieving that goal.

6. *Differences in search space.* Finally, inlining strategies may differ from one another by differences the search space. There may be different strategies based on how the search space is pruned. For instance some inlining strategies may consider the whole call graph while others only consider a section of the call graph. Some may be based on hotness, etc.

3.1 Inlining Strategies

Problem formulation. Inlining is usually formulated as a knapsack problem. Scheifler was one of the first people to reduce inlining to a knapsack problem [39] in an Ahead Of Time (AOT) compiler. Since then, reducing inlining to the knapsack problem has been the default way to solve inlining problems. Paul Berube describes the inlining strategies used by the LLVM compiler as using the greedy knapsack solution over strongly connected components in the CG [10]. Arnold et al. study of different inlining strategies that use a solution to the knapsack solution as a meta-algorithm [5]. Shankar et al. explicitly mention reducing inlining to the knapsack problem [41] in the J9 IBM JVM. However, this is not the only formulation of the problem. Chang et. al showed that the expansion of nested call sites cannot be modeled as a knapsack problem [13] due to the changes in method size of any method F once inlining of method L into method F. As such, Chang et al. [13][p 358] treat inlining as an *expansion sequence control* minimization problem. "The goal of expansion sequence control is to minimize the computation cost incurred by the expansion of [...] selected function calls." While the formulation of the inlining problem described by Chang et al. is similar to the knapsack problem, it includes another constraint on the order in which procedures should be expanded. Inlining can also be formulated similarly to the knapsack solution, however heuristics are also used in order to guarantee that call sites that meet some properties are inlined. Hazelwood et al. describe an inlining strategy that is augmented with

heuristics in the Jikes RVM [25]. While the inlining problem is often formulated as a knapsack problem, *nested inlining* is actually a more complex problem than the knapsack problem. Dean et al. [18] formulate the problem without a notion of available budget. Instead the final inlining decision happens as long as it is deemed the ratio of time savings to space cost is above a particular threshold.

Algorithmic differences. There are multiple algorithms to solve the knapsack problem. There is the traditional greedy inlining algorithm (which solves the linear relaxation on the knapsack problem). There is the dynamic-programming algorithm for solving the integer-programming problem formulation of the knapsack problem. Arnold et al. studies the greedy knapsack algorithm with different weights and values of inlining candidates [5]. Shankar et al. also use the greedy inlining algorithm [41]. Even though the greedy solution to the knapsack problem does not guarantee an optimal solution, it is often favoured to the dynamic-programming algorithm because of the lower complexity. The dynamic-programming algorithm must consider all elements that may be included in the knapsack. However, when performing nested inlining, every time a new inlining candidate is added to the set of inlined call sites, more call sites may be added to the set of call sites to consider. Even though the same is true for the greedy inlining algorithm, it is easier to modify the greedy inlining algorithm to take into account the newly created call sites. One can add the newly created call sites at the beginning of the list, at the end, or just re-sort the list. While this variation on the greedy knapsack algorithm provides no guarantees about the optimality of the results, experimentally it performs well enough.

Implementation differences. If an algorithm is vague enough to allow implementation differences, then even though the same high-level algorithm is followed, differences in implementation may exist. For example, when modeling inlining as the greedy solution to the inlining problem inlining candidates are sorted from highest to lowest benefit-cost ratio. After an inlining candidate has been chosen to be inlined, new call sites are added to the list of inlining candidates to consider. Since the traditional greedy solution to the knapsack

problem requires all elements to be known in advance, and nested inlining produces new elements after each element is selected for inlining, the traditional greedy solution to the knapsack problem does not specify where to place these newly created elements in the queue. One can consider placing them at the beginning, the end, or re-sorting the queue of inlining candidates to consider.

Goal differences. The majority of inlining strategies attempt to minimize run time of the program [5], [10], [39]. The inlining strategy proposed by Hazelwood et al. attempts to minimize run time, however due to heuristics it also guarantees that methods smaller than a size thresholds are always inlined. Dean et al. also use inlining trials to minimize run time [18]. However, some inlining strategies may attempt to minimize or maximize other program properties. For example, Appel describes rules for inlining that minimizes the size of the program representation [4]. Shankar et al. design an inlining strategy that minimizes *object churn* (i.e., the excessive creation of short lived objects) [41]. Sewe et al. extend the Jikes RVM inlining strategy to take into account optimizations that will take place after inlining has happened [40].

Goal estimate differences. The way the impact of inlining on the optimization criteria (or the goal) is estimated may differ. For example, there are a lot of inlining strategies that attempt to minimize run time. Scheifler estimates the impact of inlining on run time by looking at offline profile information and estimating the number of dynamic calls from the offline profile information. [39]. The use of offline profile information is popular for AOT compilation. Paul Berube extends offline profile information by combining profile information from multiple inputs [10]. Arnold et al. describe several ways to estimate the impact of inlining on reducing the number of dynamic calls [5]. For example, the nodes in the call graph may be annotated with *method invocation counters*, or the edges in the call graph may be annotated with *call counters*. Method invocation counters are counters placed in instrumented methods that increment every time the method has been called. These do not preserve context sensitivity as the counter is increased independently of which method is the caller. The call counters differ from method invocation counters in that call counters allow to determine how often a method is called from a specific call

site. Hazelwood et al. maintain records to preserve more context on the method invocation counters [25]. Dean et al. determine the number of instructions saved from optimizations by estimating how many instructions are saved by an optimization and multiply it with their expected execution frequency [18]. As these examples illustrate, there are many different ways of estimating a program property.

Differences in search space. There may be different heuristics used to prune the search space, thus leading to only exploring a subset of all possible inlining decisions. Some of the ways in which the search space is pruned is by using heuristics based on the hotness of a method. If a method is cold and does not reach a hotness threshold, then neither that method, nor its descendants, will be considered for inlining. This restriction may lead to some suboptimal inlining decisions in cases where descendants of the cold method have loops that are repeatedly executed.

3.2 Different Types of Analyses

The previous section describes all the different ways in which an analysis may influence inlining. However, we also need to compare the analysis themselves. There are two approaches to interprocedural data-flow analysis. The first may be called *functional approach* [42, Chapter 7], *method summary approach* [12], [32], or *bottom-up* [47] analyses. The second may be called k-CFA [43], *call-strings* approach [42, Chapter 7], or *top-down* analysis [47].

The call-strings approach to interprocedural data-flow analyses can be seen as an immediate extension of intraprocedural data-flow, where the control flow graph has been augmented to include edges between call sites and beginning of procedure, and return statements to call sites. The call-string approach implies that every time a call site is encountered, the callee has to be reanalyzed under the new context. To improve the efficiency of this interprocedural analysis, sometimes analyses are limited to k contexts. Limiting analyses to k contexts is called k-limiting. While k-limiting produces results, the analysis' running time is exponential for large programs.

In the functional approach, summary flow functions are computed for each procedure. These summary flow functions are used as the flow function for the entire call block. The flow functions need to be distributive and close under composition. IFDS/IDE are two well-known examples of interprocedural analysis frameworks done in the functional approach [35], [38] The functional approach, tends to be faster because functions are not reanalyzed.

One can think of the call-strings analysis as a function that takes a program and some inputs and produces results. Meanwhile, the functional approach is a function that takes a program and produces a function that takes inputs and produces results. In other words, the functional approach is just a curried version of the call-strings analysis. Method summaries produced by the functional approach are the functions returned by the functional approach.

$$\textit{call-strings analysis} : \textit{program} \times \textit{inputs} \rightarrow \textit{results}$$

$$\textit{functional approach} : \textit{program} \rightarrow (\textit{inputs} \rightarrow \textit{results})$$

Chapter 4

IDT-Based Inliner

To use the procedure described in by Craik et al., we need to provide an IDT and annotate it with weights and values corresponding to the cost and benefit of a given inlining candidate [16]. The value used for the cost of inlining should reflect the compilation time. There are several different metrics that serve as an approximation to compilation time (e.g., nodes in the AST, arithmetic operations, number of instructions). We use the number of bytecodes in a method as the cost of inlining. The benefit of inlining should be a function of how often a method is executed and how optimized a method will be to its calling context. That is, the benefit of inlining is a function of the *direct benefits* and the *indirect benefits* of inlining. Direct benefits will be covered more extensively in Section 4.2 and indirect benefits will be covered in Chapters 5 and 6

Figure 4.1 describes the proposed inlining strategy in pseudocode. The *generateIDT* function on Line 2 builds the IDT. The cost and the *direct benefits* of inlining are calculated during the generation of the IDT.

The next step is to compute the *indirect benefits* of inlining. The indirect benefits of inlining are calculated using a data structure called *method summary* and through the static approximation of arguments' run-time values. The method summaries encode constraints to be satisfied for potential optimization opportunities. The static approximation to arguments' run-time values are used to see if the constraints are satisfied.

Line 9 of Figure 4.1 shows a call that takes a target method as an input and

```
Require: method is method requested for compilation by the VM
Require: budget is inlining budget for method
Ensure: ∀node ∈ IDT, node has benefit > 0 and methodSummary exists
 1: procedure INLINER(method, budget)
 2:     IDT ← GENERATEIDT(method, budget)
 3:     for each node ∈ DFS(IDT) do
 4:         argumentEstimates ← ESTIMATEARGUMENTSTOCALLSITESIN(node)
 5:         for each target ∈ node do
 6:             if target.methodSummary then
 7:                 methodSummary ← target.methodSummary
 8:             else
 9:                 methodSummary ← GENERATEMETHODSUMMARY(target)
10:             end if
11:             target.benefit ← METHODSUMMARY(argumentEstimates)
12:         end for
13:     end for
14: end procedure
```

Figure 4.1: Main algorithm of the proposed inliner.

generates a method summary. The potential optimization opportunities are dependent on the arguments and the contents of the method being summarized. The process of creating an entry in the method summary is the task of the compiler developer. The prototype described in this thesis includes several optimizations in the method summary. These optimizations and their encoding into a method summary are described in Chapter 6.

Method summaries need an estimate of the argument values to compute which optimizations will take place. If the estimate of the arguments satisfies the constraints, then the optimization is realizable and only depends upon the function being inlined. The abstract interpreter estimates the run-time values held by the arguments. Line 4 calls the method to estimate the values held by each argument at call sites. The abstract semantics and the transfer functions are detailed in Chapter 5.

The static benefit to be assigned to each inlining candidate is computed after the argument estimates and method summaries. This static benefit is the aggregation of individual weights assigned to different realizable optimizations. After the IDT is annotated with the benefit value, the inliner uses the knapsack

algorithm described by Craik et al [16]. This process computes the set of nodes in the IDT and, given a constraint on a budget, maximizes the benefit of the nodes in the set.

4.1 Building an Inlining Dependency Tree

The IDT is constructed during the inlining pass. The inlining pass receives, as arguments, the method requested for compilation (i.e., the *compilation request*) and a *budget*. The budget constrains inlining to a maximum increase in bytecode count. The algorithm used for constructing the IDT in Figure 4.2. The `generateIDT` procedure receives the compilation request, the budget, and a root node that holds the compilation request as arguments. The construction of the IDT is a recursive process that halts when the inlining budget has been exhausted.

In Figure 4.2 the control flow graph of a method is accessed through the following syntax: *method.cfg*. The cost of inlining (which is reflected by updating the budget on Line 12 and Line 14) is the size of the target method in number of bytecodes. This is because the size of a method is often a good indication of how long a method will take to compile as different analyses used during compilation have a super-linear time complexity on the size of code [10].

After checking for the stopping condition on Line 2, the algorithm iterates over each basic block on the argument *method*. Each instruction of each basic-block is inspected to determine whether it is a call site. If a call site is found, the targets of the call site are resolved through the method `findCallSiteTargets()` available in OpenJ9. On line 8, the pseudocode shows the resolution of targets through `findCallSiteTargets()`. Because of dynamic loading and dynamic dispatch, it is impossible to determine all potential targets of a call site. However, `findCallSiteTargets()` provides a list of targets for interface call sites and virtual call sites that have a high likelihood of being called.

The method `findCallSiteTargets()` may use method invocation counts, argument pre-existence information and other analyses to resolve indirect call sites. In the prototype, the IDT allows for a single target of indirect call sites

```
1: procedure GENERATEIDT(method, budget, root)
2:     if budget < 0 then
3:         return
4:     end if
5:     for each basicBlock ∈ REVERSEPOSTORDER(method.cfg) do
6:         for each instruction ∈ basicBlock do
7:             if instruction = invocation then
8:                 targets = FINDCALLSITETARGET(method, instruction)
9:                 for each target ∈ targets do
10:                    node = new Node(target)
11:                    INSERT(root, node)
12:                    budget = budget − SIZE(target)
13:                    GENERATEIDT(target, budget, node)
14:                    budget = budget + SIZE(target)
15:                end for
16:            end if
17:        end for
18:    end for
19: end procedure
```

Figure 4.2: Generating the IDT

to be considered; however, there is no real restriction on how many targets may be added per call site. The decision about how many targets should be added per call site must consider the trade-off between analysis time and likelihood that considering more targets improves the inlining plans. This trade-off is likely a function of how monomorphic the call site is and the depth at which the call site is found on the IDT.

Similar to the default inlining strategies in OpenJ9, in our implementation, `findCallSiteTargets()` does not attempt to resolve methods that have never been called. This design decision is an example of pruning the search space during the inlining process, because inlining methods that have never been called is likely to be a bad decision. Inlining relies on the principle that past behaviour is an indication of future behaviour. Thus a method that has never been visited implies that a method is likely to not be executed in the future. If a method is likely to not be executed in the future, then inlining it will not have any direct benefits.

A new node in the IDT is constructed after a target for a call site is found.

This node holds all the necessary information that describes this call site. This node becomes a child to the node holding the current method. The algorithm updates the budget on Line 12 by subtracting the size of the target. The size of the target is used as an estimate for the cost of inlining. Each target then undergoes the same process recursively until the budget is consumed.

When the inspection of a method is finished, the algorithm adds the size of the target to the budget as seen on Line 14. Resetting the budget to its previous value ensures that each branch of the IDT, from the root node to the leaves, holds methods whose sizes do not exceed the budget. The nodes in the IDT correspond to the universe of possible inlining decisions.

4.2 Dynamic Inlining Benefits

The algorithm annotates nodes in the IDT with weights that correspond to the *direct benefits* of inlining. It also computes the *call ratio*, a value to represent the direct benefits of inlining. The call ratio estimates how often the program invokes a method upon executing the method at the root node of the IDT. If a method is inside a loop, its call ratio has a value higher than one, if the method is inside a conditional block, then its call ratio will be less than one, and if the method is unconditionally executed with no loops, its call ratio estimate is 1.

The value of *callRatio*(m_{root}) is defined axiomatically as 1. The algorithm computes the basic-block frequency using online profile information provided by the profiling infrastructure in OpenJ9. Generally, the root method contains n basic-blocks. We use the notation *frequency*(bb_i^m) to denote the frequency of basic-block i in method m. The frequency is a value scaled from 0 to 10,000. Where a value of 0 indicates that a block has never been visited and a value of 10,000 indicates that the basic-block is the most visited basic-block in the method. The reason the value is caled from 0 to 10,000 is for historical reasons. The basic block profiler in the JIT compiler available in OpenJ9 scaled frequencies to 10,000.

To compute the call ratio for any method m_{callee} with respect to the method m_{caller}, the algorithm queries for the entry basic-block in method in m_{caller}

(bb^{caller}_{entry}), and the basic-block containing the call site ($bb^{caller}_{call\ site\ to\ callee}$). Next, the algorithm queries for the block frequency from OpenJ9's profiling infrastructure. The call ratio for a method m_{callee} with respect to a caller m_{caller} is computed via the following formula.

$$callRatioCallerCallee(m_{caller}, m_{callee}) = \frac{frequency(bb^{m_{caller}}_{call\ site\ to\ m_{callee}})}{frequency(bb^{m_{caller}}_{entry})} \quad (4.1)$$

We also extend the call ratio caller-callee formula to account for virtual methods. The *targetPercentage* field is an estimate of how often method m_{callee} is the target of the call obtained from its call site. The profiling framework for OpenJ9 provides the values of *targetPercentage*, which depending on the profiling infrastructure (chosen at run-time) may be either context insensitive or context sensitive. A value of 1 indicates that m_{callee} is the sole target of the call site and a value of 0 indicates that it has never been called from that call site.

$$callRatioCallerCallee(m_{caller}, m_{callee}) = \frac{frequency(bb^{m_{caller}}_{call\ site\ to\ m_{callee}})}{frequency(bb^{caller}_{entry})} \times m_{callee}.targetPercentage \quad (4.2)$$

This formula works well to compute the call ratio between caller and callee. However, we are interested in computing the call ratio between any node in the IDT and the root method. That is, $callRatio(m_{root}, m_c)$ is computed by the following formula.

$$callRatio(m_{root}, m_c) = \prod_{\forall m_\alpha \in ancestors(m_c) \cup \{m_c\} - m_{root}} callRatioCallerCallee(parent(m_\alpha), m_\alpha) \quad (4.3)$$

Which can also be re-written as:

$$callRatio(m_{root}, m_c) = callRatio(m_{root}, parent(m_c)) \times callRatioCallerCallee(parent(m_c), m_c) \quad (4.4)$$

This formula allows the ancestors' call ratios to contribute to the descendant's call ratio. Specifically, the *callRatioCallerCallee(parent(m_c),m_c)* value is scaled by *parent(m_c)*'s call ratio. This is a desirable property of the formula because the invocation frequency of a method depends on the execution frequency of its parent.

The calculation of call ratios is done during the construction of the IDT to save time. Call ratios are stored as doubles during the construction of the IDT. However, once the algorithm finishes building the IDT annotated with call ratios, call ratios are scaled to integers between the values of 1 and 10,000. The reason why call ratios are scaled is to mimic OpenJ9's profiling infrastructure which uses values between 1 and 10,000 to denote basic-blocks' frequencies.

Chapter 5

Estimating Run-Time Argument Values

We have already talked about the types and values available in the JVM in Section 2.1.2 and how the abstract interpretation framework allows for a safe estimate of run-time values of a program in Section 2.3. In Section 2.2.1, we also reviewed the `VPConstraint` framework available in OpenJ9 and how it serves as an abstraction of the concrete types found in the JVM. Those sections serve as the basis for this chapter in which we will discuss our concrete implementation of an abstract interpreter and its differences with the traditional abstract interpretation framework.

In Section 5.1, we specify the handling of the calls. In Section 5.2, we specify the flow of control used in the abstract interpreter. We show how the abstract state is transferred from different points in the program as inputs to the transfer functions. We also define how we merge abstracts states within the program. In Section 5.3, we summarize the abstract semantics for all the bytecodes available in the JVM.

5.1 Call Stack

At the beginning of a method's abstract interpretation (i.e., when calling method estimateArgumentsToCallsitesIn() in Figure 4.1), the abstract frame loads the root method, and the analysis loads the abstract local variable array with an estimate of the call site's arguments. The abstract interpretation

of the root method differs from the rest of the nodes in the IDT, because, unlike the rest of the nodes, the root method has no parent node in the IDT. Having no calling context for the root method means that the abstract arguments passed onto the root method contain as much static information as the formal arguments in the root method signature. All other nodes in the IDT have the abstract arguments from the operand stack passed as arguments. The root method then undergoes abstract interpretation the same way the rest of the nodes as discussed in Section 5.3.

Visiting a node n on the IDT is equivalent to performing abstract interpretation on the method m stored in the node n. During the abstract interpretation of method m_{caller}, if a call site c is found, the targets of the call site are no longer resolved with `findCallSiteTargets`. Instead, because the target methods are already encoded in the IDT as child nodes, the call site c is resolved by looking at the children of node n. There can be multiple targets t for a single call site c, so we adopt the notation t_i to distinguish individual targets of a call site. The order in which multiple targets t are abstractly interpreted is undefined. After picking a target to abstractly interpret, an abstract frame $\hat{f}$ is created and placed on top of the abstract call stack.

Each abstract frame $\hat{f}$ contains an abstract operand stack $\hat{s}$ and an abstract variable array $\hat{a}$. Upon encountering a call site c to a method m_{callee}, the elements in the abstract operand stack $\hat{s}_{caller}$ of the abstract frame $\hat{f}_{caller}$ correspond to the arguments to method m_{callee}. These abstract arguments are placed in the local abstract variable array $\hat{a}_{callee}$ of the abstract frame $\hat{f}_{callee}$ corresponding to the method m_{callee} and an empty abstract operand stack $\hat{s}_{callee}$ is initialized in abstract frame $\hat{f}_{callee}$. The placement of the arguments in the caller's stack to the callee's array matches the concrete semantics specified by the Java Virtual Machine Specification [31].

After the creation of the abstract frame $\hat{f}_{callee}$ the abstract interpreter proceeds as follows: halt the interpretation of method m_{caller} in frame $\hat{f}_{caller}$; abstractly interpret method m_{callee} in frame $\hat{f}_{callee}$; recursively interpret methods that are called; Upon finishing interpreting method m_{callee}, pop the the frame f_{callee} off the abstract call stack and continue analyzing method m_{caller}.

When call-sites in leaf nodes are visited the abstract interpreter safely approximates the return values by placing $\top$ in the stack if the method returns values. If no return value is expected, then nothing is placed on the stack. The process continues until all nodes in the IDT are visited. If method m_{callee}'s signature indicates a return value, the abstract interpreter places a safe approximation based on the function's signature on the operand stack $\hat{s}_{caller}$.

Our current implementation limits the amount of information transferred from caller to callee. When the interpreter finishes analyzing method m_{callee}, no information except the return value's type (as specified by the method's signature) may be transferred to m_{caller}. Future work includes extending the abstract interpreter to allow for more precise estimates to be returned from call sites. For example, the type returned by a method may be a class derived from the method signature's class. Obtaining the return value's type from the method's signature is less precise than inspecting the type of the value at the return statements. Optimizations that depend on more precise estimates of return values will likely not satisfy the constraints needed to guarantee that the optimization will take place.

5.2 Control Flow

The analysis interprets the basic blocks in reverse post-order. This order ensures that in the absence of cycles, when interpreting node n, all predecessors of node n have been interpreted before n is interpreted. During the interpretation, the abstract operand stack, the abstract variable array, and the abstract call stack are maintained according to the abstract semantics. The abstract state of the program is completely determined by the abstract variable array, abstract operand stack and the abstract call stack.

At the beginning of the interpretation of basic block x, the abstract state must be transferred from the *directPredecessors*(x) (according to the control flow graph) to x to be interpreted. At the end of the interpretation of x, the abstract state is stored and can be accessed through x. This stored abstract state will be used as an input for the successors of x. The abstract state will be

used together with instructions to compute the next abstract state according to the transfer function and the abstract semantics. Because reverse post-order is used to iterate over the basic blocks, we have to consider the following cases:

Case 1 $|directPredecessors(x)| = 1$ and $directPredecessor(x) = y$: y has already been interpreted and has an abstract state stored.

Case 2 $|directPredecessors(x)| > 1$ and $|backEdges(x)| = 0$ and $directPredecessors(x) = Y$: all nodes in Y have been interpreted and thus have an abstract state stored.

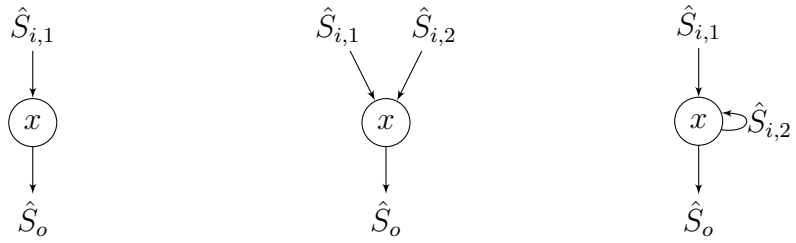
Case 3 $|directPredecessors(x)| > 1$ and $|backEdges(x)| > 0$ and $backEdges(x) = Z$: none of the nodes in Z have been interpreted and thus have no abstract state stored.

These three cases are shown pictorially in Figure 5.1.

For Case 1, the final abstract state found in y needs to be used as an input to the flow function used to interpret the beginning of x. For Case 2, the final abstract states found in nodes in Y need to be merged and used as an input to the flow function used to interpret the beginning of x. For Case 3, the final abstract states in nodes in Y can be known, however the abstract states in nodes in Z remain unknown.

In a traditional abstract interpreter, when x is encountered for the first time, only the abstract states in Y are considered as input for x. The output abstract state of x is then propagated to its successors. In traditional abstract interpretation, in subsequent interpretations of x, the nodes in Z would then have an abstract state, and would be included as input to x. Intermediate abstract states, those states that are computed before reaching the least fixed point, would be considered unsafe estimates to run-time values. Only after the abstract interpreter finds the least fixed point, are the abstract states safe to use for analysis.

Our implementation of abstract interpretation differs from the traditional case in that nodes are only interpreted once. This design decision is made on the basis that JIT compilation needs to be done fast and iterating until a



(a) Input abstract state $\hat{S}_{i,1}$ is transferred to block x and x produces output abstract state $\hat{S}_o$.

(b) Input abstract state $\hat{S}_{i,1}$ and $\hat{S}_{i,2}$ is transferred to block x and x produces output abstract state $\hat{S}_o$.

(c) Input abstract state $\hat{S}_{i,1}$ and $\hat{S}_{i,2}$ is transferred to block x and x produces output abstract state $\hat{S}_o$. However, $\hat{S}_{i,2}$ is a back edge so there is no abstract state to be transferred.

Figure 5.1: Different cases of input abstract state transmission.

fixed point is reached may be too costly. This decision however has the impact that our implementation is would not be considered safe. In order to maintain safety, we decrease the precision of our analysis. Instead of iterating over the basic blocks until fixed point is reached, our abstract interpreter upon seeing Case 3 will immediately discard the abstract state from Y and assume $\top_{\hat{S}}$ for the abstract state used as input. $\top_{\hat{S}}$ is defined as replacing all values in the abstract operand stack $\hat{s}$ and abstract variable array $\hat{a}$ with $\top$ elements. This jump in the lattice from $\hat{S}$ to $\top_{\hat{S}}$ reduces precision but maintains safety.

A possible extension for the current implementation would be to switch between single iteration and iteration until fixed point is reached. Depending on the precision of the analysis requested and the budget allocated for analysis, it would be possible to decide how long to execute the analysis at run-time. If a more precise analysis is requested, then the number of iterations could be increased. This extension requires a non-trivial modification to the current implementation.

We define the $\sqcup_{\hat{S}}$ operator as a binary operator between two abstract states $\hat{S}_i$ and $\hat{S}_j$. The $\sqcup_{\hat{s}}$ operator performs the least upper bound operation for the abstract operand stacks $\hat{s}_i$ and $\hat{s}_j$, and the abstract variable arrays $\hat{a}_i$ and $\hat{a}_j$. We define the $\sqcup_{\hat{s}}$ and $\sqcup_{\hat{a}}$ operator for the abstract operand stack and the

```
1: procedure ⊔_Ŝ(Ŝ_i, Ŝ_j)
2:     returnState ← new Ŝ
3:     if Case 1 then
4:         returnState.stack ← Ŝ_i.stack
5:         returnState.array ← Ŝ_i.array
6:     else if Case 2 then
7:         returnState.stack ← Ŝ_i.stack ⊔_ŝ Ŝ_j.stack
8:         returnState.array ← Ŝ_i.array ⊔_â Ŝ_j.array
9:     else
10:        returnState.stack ← ⊤_ŝ
11:        returnState.array ← ⊤_â
12:    end if
13:    return returnState
14: end procedure
```

Figure 5.2: The computation of $\sqcup_{\hat{S}}$ for abstract states.

```
1: procedure ⊔_ŝ(ŝ_i, ŝ_j)
2:     returnStack ← new ŝ
3:     for each ê_i, ê_j ∈ZIP( ŝ_i, ŝ_j) do
4:         ê_res ← ê_i ⊔_ê ê_j
5:         returnStack.push(ê_res)
6:     end for
7:     return returnStack
8: end procedure
```

Figure 5.3: The computation of $\sqcup_{\hat{s}}$ for abstract stacks.

abstract variable array. These procedures are described in Figures 5.2, 5.3, and 5.4.

The $\sqcup_{\hat{e}}$ operator for each abstract element is defined in terms of the `merge` function as found in the `VPConstraint` framework. The `merge` function computes the least upper bound operation for the different classes modeled by `VPConstraint`. For example, the `merge` operator for the `VPClass` constraint is defined as the first common ancestor in the class hierarchy between the two classes being merged. Similarly the $\sqcap_{\hat{e}}$ operator for each abstract element is defined in terms of the `intersect` function as found in the `VPConstraint` framework. However, instead $\sqcap_{\hat{e}}$ computes the greatest lower bound.

```
1: procedure ⊔_â(â_i, â_j)
2:     returnArray ← new â
3:     for each ê_i, ê_j ∈ZIP( ŝ_i, ŝ_j) do
4:         ê_res ← ê_i ⊔_ê ê_j
5:         returnArray.pushBack(ê_res)
6:     end for
7:     return returnArray
8: end procedure
```

Figure 5.4: The computation of $\sqcup_{\hat{a}}$ for abstract arrays.

5.3 Abstract Semantics

To show the effect of instructions on the abstract operand stack, the abstract frame, and the abstract variable array, we define the abstract semantics in a similar way to the Java Virtual Machine Specification [31, Chapter 6]. However, instead of listing all bytecodes, Table A.1 groups the instructions encountered in the JVM. The first column contains the name we use to refer to a group of similar instructions. The second column contains the mnemonic name for the instructions in the JVM.

The description of the semantics is in Appendix A. Please note that the semantics for CALLSTACK and CONTROLFLOW are more thoroughly explained in Sections 5.1 and 5.2.

5.3.1 Relating Argument Estimates to Call Sites

The calling convention for the JVM specifies that arguments must be placed on the operand stack before making a call. These semantics almost allow for the safe estimation of run time values placed on the stack. Unsafe values come from the *invokedynamic* instruction. Functions which are not resolved at analysis time can be produced due to *invokedynamic*. These functions use variadic arguments. As such, in order to maintain safety, the analysis would need to determine how many arguments are passed to the unresolved function. Future work will address this source of unsafety.

Let's consider the simple example shown in Figure 5.5.

The objective of the analysis is to find the contents of the stack before

```
1  public static void example(int);
2    Code:
3       iconst_0
4       istore_1
5       iload_0
6       ifge         11
7
8       iconst_1
9       istore_1
10      goto         14
11
12      bipush       100
13      istore_1
14
15      new          #2       // class DerivedClass
16      dup
17      invokespecial #3      // Method DerivedClass."<init>":()V
18      astore_2
19      aload_2
20      iload_1
21      iload_0
22      invokestatic #4       // Method foo:(LBase;II)V
23      return
```

Figure 5.5: Example to illustrate abstract interpretation

Table 5.1: Values in abstract array at different line numbers

Line Number	**Array Index**	**Content**
Line 6	0	$\top$
Line 10	0	$\top$
Line 10	1	1
Line 12	0	$\top$
Line 12	1	100
Line 15	0	$\top$
Line 15	1	[1, 100]
Line 22	0	$\top$
Line 22	1	[1, 100]
Line 22	2	`DerivedClass`

Top of stack
$\top$
$[1, 100]$
`DerivedClass`
Bottom of stack

Figure 5.6: Abstract stack containing abstract argument estimates. `DerivedClass` is known to be not null because its provenance is from the instruction `new`.

executing Line 22. In order to do so, the semantics specified previously model the stack and the abstract array. On Line 6 the contents of the variable array are shown in Table 5.1. The stack at position Line 6 is empty. This abstract state is transferred to the basic blocks starting at on Line 8 and Line 12.

The abstract variable array on Line 10 is shown in Table 5.1. Similarly, the abstract variable array on Line 13 is shown in Table 5.1. The stacks remain empty at these lines. These two states will be merged before Line 15 executes. The merged states resulting on the abstract state shown in Table 5.1.

Abstractly interpreting the rest of the instructions starting at Line 15 until Line 22 should result in the abstract stack as shown on Figure 5.6 and the abstract variable array on Table 5.1.

Chapter 6

Determining Possible Optimizations

We use abstract interpretation as described in Chapter 5 to determine an estimate to the run-time values of arguments. These argument estimates are used by a data structure named method summary. This chapter describes the method summary and how we encoded several analyses in it. The method summary contains logical predicates that reflect the optimizations done by the compiler and the conditions on which those optimizations are applied. It is the job of the compiler developer to determine that the construction of such predicates correctly reflects the potential optimizations that can be applied by the compiler due to inlining.

We have developed a procedure in order to construct predicates that reflect the following potential optimizations:

1. branch folding
2. null checking folding
3. cast folding
4. instance of folding
5. partial evaluation

Figure 6.1 contains the pseudo-code for our algorithm to generate method summaries. Our procedure involves finding the uses of argument definitions.

```
1: procedure GENERATEMETHODSUMMARY(method)
2:     for each argument ∈ method do
3:         for each use ∈ Use(argument) do
4:             if use ∈ OptimizableCode then
5:                 UPDATE(method.methodSummary)
6:             end if
7:         end for
8:     end for
9: end procedure
```

Figure 6.1: Pseudocode for generating method summaries

```
1  public static boolean branchfolding(boolean);
2    Code:
3       0: iload_0
4       1: ifeq 6
5       4: iconst_1
6       5: ireturn
7       6: iconst_0
8       7: ireturn
```

Figure 6.2: Example code to show branch folding constraints

Potential optimizations are found when arguments are used as operands to instructions that are capable of being eliminated. A *potential optimization* is a code pattern that may enable optimizations. The optimizations may not necessarily be realizable, because they may depend on the estimates of the argument values. The potential optimization is encoded into the method summary as a constraint over the possible values held by the arguments.

Figure 6.2 shows the bytecode of a method named `branchfolding()`. This method takes a single boolean argument and loads it onto the concrete operand stack. Depending on the argument's concrete value, the method will either return the value 1 or 0.

Iterating over the basic blocks of `branchfolding()` in reverse post-order means that we start analyzing the instructions located at byte offset 0 and 1 of method `branchfolding()`. The JVM Specification [31, Section 3.6] states that upon the start of a method's execution, the arguments are placed in the local variable array. First, the value of the local variable array at position zero is

pushed onto the stack. Because no instruction has overwritten the value of the local variable array at position zero, it is safe to say that `0: iload_0` loads an argument onto the stack. Instruction `1: ifeq` is a branching statement that is conditional on the argument. At that point, the analysis does not know anything about the potential values of the argument; however, due to the semantics of the `ifeq` instruction, it is safe to say that if the argument is less than or equal to zero, then the bytecode in the sixth position (i.e., the instruction `6: iconst_0`) will be the target of the branch. Otherwise, the bytecode in the fourth position (i.e., the instruction `4: iconst_1`) will the target of the branch.

When the analysis first encounter `1: ifeq`, it performs a check to determine whether the value on the operand stack is a direct use of an argument. If the check succeeds, we determine which argument was used. The JVM Specification [31, Section 3.6] states that the arguments are placed in order in the local variable array upon starting the execution of a method. Since the value on the operand stack is obtained from the instruction `iload_0`, we know that the first argument is the one used as a branch test. At that point, GENERATEMETHODSUMMARY knows that the test is performed on an argument, and which argument is being tested.

At bytecode `1: ifeq`, a potential opportunity for branch folding exists if the first argument is equal to zero. We also say that there is another opportunity for branch folding at `1: ifeq` if the zeroth argument is not equal to zero. The method summary encodes the use of the argument and the branch inequality as a constraint that must be met by the argument value estimates. Generally speaking, if the constraints are met by the argument estimates, then the optimization can take place.

Concretely, each potential optimization opportunity is encoded as a row in a table where each column represents arguments. Each cell under an argument represents a constraint on that argument in order for the optimization to be realizable. Table 6.1 shows an example of a minimal method summary constructed for the example code in Figure 6.2.

The abstractions in `VPConstraints` has some limitations. We cannot

Table 6.1: Method summary for Figure 6.2

Potential Optimization	**Bytecode**	**Argument 0**
Branch Folding →	`1 : ifeq`	[`INT_MIN`, −1]
Branch Folding ←	`1 : ifeq`	[0, 0]
Branch Folding →	`1 : ifeq`	[1,`INT_MAX`]

express the range [`INT_MIN`, −1]∪[1, `INT_MAX`] in the `VPConstraints` framework, so separating it into to separate disjoint ranges is necessary. The method summary overcomes this limitation by allowing multiple rows to encode the same constraint. For example, the range [`INT_MIN`, −1] ∪ [1, `INT_MAX`] will be expressed by two different entries in the method summary. One containing the constraint [`INT_MIN`, −1] and the other containing [1, `INT_MAX`]. Additionally, multiple potential optimizations may be encoded in the same method summary. We outline other optimizations considered in the following sections. Ranges constraints that model intersection (i.e., [`INT_MIN`, 0] ∩ [−1, 0]) on a single argument are modeled implicitly by their using the result constraint in the method summary. Modeling the and/intersection across arguments is possible by placing constraints on different cells. In other words, a method summary can be read as a set of boolean functions (one for each row), where all elements in a non-empty cell are *and-ed* with each other. If a boolean function evaluates to true, it means that an optimization has been found.

6.1 Computing Constant String Length

From literal strings, it is possible to substitute a call to the method `String.length()` by a compile time constant. Therefore, upon encountering calls `String.length()`, a row on a method summary is created. Figure 6.3 shows that the bytecode for the method `wrapperStringLength()`. On Line 4 the first argument is used as an implicit argument to the method `String.length()`. To determine whether the length of the argument can be known at compile time, we must inspect the type of constraint.

```
1  public static void wrapperStringLength(java.lang.String);
2    Code:
3       0: aload_0
4       1: invokevirtual #2                  // Method java/lang/String.
           length:()I
5       4: pop
6       5: return
```

Figure 6.3: Example code to show string length constraints

Table 6.2: Method summary for Figure 6.3

Potential Optimization	**Bytecode**	**Argument 0**
String Length	`1 : invokevirtual`	`asConstString`

If the argument estimate is encoded in a `VPConstString` constraint, then we know that the length of the argument string can be known at compile time. The method summary produced upon visiting the `wrapperStringLength` method is found on Table 6.2. The `asConstString()` function is defined in the `VPConstraint` framework in such a way that if the constraint is not derived from the `TR::VPConstString` then it returns `false`. As such, when the function `asConstString()` is applied on the estimate of argument, it must return `true` for the call to `String.length()` to be folded.

6.2 Null-Check Folding

A variable containing a reference to `null` may produce run-time exceptions. As such, it is important to perform a check to verify that the reference does not point to `null` before an operation is performed on the variable. There are several ways to check for the null-ness of a variable [2]. One of such ways is to invoke the `getClass()` method on the variable to test. Calling `getClass()` on an non-`null` object allows the compiler to avoid further `null` checks on the same object. This is because the compiler is able to assume that if the call to `getClass()` fails, an exception is thrown and instructions that execute after `getClass()` are never executed. However, if the call to `getClass()` succeeds,

```
1  public static void nullCheck(java.lang.Object);
2    Code:
3       0: aload_0
4       1: invokevirtual #2                  // Method java/lang/Object.
           getClass:()Ljava/lang/Class;
5       4: pop
6       5: return
```

Figure 6.4: Example code to show null check constraints

Table 6.3: Method summary for Figure 6.4

Potential Optimization	**Bytecode**	**Argument 0**
Null Check	`1 : invokevirtual`	`isNullObject`
Null Check	`1 : invokevirtual`	`isNonNullObject`

the instructions that execute after `getClass()` are allowed to assume that the object which called it is non-`null`.

The method summary in Table 6.3 is very similar to the method summary in Table 6.2. However, in this case, instead of checking the argument's type, we will check for the argument's nullness. If the argument is estimated to be `null`, then the code should be replaced to always raise a `NullPointerException`. If the abstract interpretation estimates the argument to never be `null`, then the compiler can eliminate the `null` check. Table 6.3 provides the method summary that is generated upon analyzing the method in Figure 6.4.

6.3 Instance Of Checking

The instruction *instanceof* can be folded away if the operand is known (or can be estimated) at compile time. As such, upon encountering the instruction, our analysis creates a new method summary entry.

We extended the `VPConstraint` framework to allow for a subset operation. The subset operation returns `true` if the second operand is a class assignable to the first operand. The subset operation allows us to determine whether the argument is a reference to class `Example` or a derived class. This definition

```
1  public static boolean instanceofCheck(java.lang.Object);
2    Code:
3       0: aload_0
4       1: instanceof   #2                 // class Example
5       4: ifeq         9
6       7: iconst_1
7       8: ireturn
8       9: iconst_0
9      10: ireturn
```

Figure 6.5: Example code to show check cast constraints

Table 6.4: Method summary for Figure 6.5

Potential Optimization	**Bytecode**	**Argument 0**
instanceof	`1 : instanceof`	`subset(Example)`

of subset corresponds to the concrete semantics of the *instanceof* instruction. Therefore, if the constraint is satisfied, the instruction *instanceof* can be folded at compilation time.

6.4 Cast Folding

A *checkcast* instruction can be folded in a similar way to the *instanceof* instruction. However instead of using the `subset` method, we use the method already available `mustBeEqual`. This corresponds to the concrete semantics of the *checkcast* instruction. The example method summary found in Table 6.5 corresponds to the example code found in Figure 6.6.

6.5 Partial Evaluation

The analysis that we developed to encode partial evaluation into the method summary is limited. First, only arithmetic instructions performed over the integer data type are taken into account. Second, only when the values estimated on the call site are members of the `VPIntConst` class are the optimizations encoded.

```
1  public static int checkCast(java.lang.Object);
2    Code:
3       0: aload_0
4       1: checkcast    #2                  // class Example
5       4: ifnull       9
6       7: iconst_1
7       8: ireturn
8       9: iconst_0
9      10: ireturn
```

Figure 6.6: Example code to show check cast constraints

Table 6.5: Method summary for Figure 6.6

Potential Optimization	**Bytecode**	**Argument 0**
checkcast	`1 : instanceof`	`subset(Example)`

Only the arithmetic instructions performed over the integers are taken into account because in the `VPConstraint` framework only the integer values can be modeled as constants in the `VPIntConst` class. The `VPIntConstraint` is the base class of `VPIntConst` or `VPIntRange`. Similar base classes exist for short and long types. The difference between `VPIntConst` and `VPIntRange` is that `VPIntConst` is a single value, which `VPIntRange` models an unknown value as a range of integers.

Partial evaluation can only take place if the argument estimates are members of the `VPIntConst` class. This is because if the values estimated from the call site are members of `VPIntRange` there is no certainty on which of the values in the range will be used during partial evaluation. What is currently being done to generate the method summary is, we keep track of the type of constraints in the abstract operand stack, and when we encounter an arithmetic operation two, checks are done. The first one to check if either operand is an argument. The second one to check if is the non-argument operand is a `VPIntConst` constraint.

Let's take a look at the example shown in Figure 6.7. In bytecode `0 : iconst_1`, we obtain a `VPIntConst` constraint from loading an immediate value

```
1  public static int partialEvaluation(int);
2    Code:
3       0: iconst_1
4       1: iload_0
5       2: iadd
6       3: ireturn
```

Figure 6.7: Example code to show partial evaluation constraints

Table 6.6: Method summary for Figure 6.7

Potential Optimization	**Bytecode**	**Argument 0**
Partial evaluation	`2 : iadd`	`asConstInt`

of 1. In bytecode `1 : iload_0` the interpreter loads a value from the argument. In bytecode `2 : iadd` both checks succeed. Then the interpreter places an entry in the method summary as seen in Table 6.6. The constraint `asConstInt` is placed in the method summary because the argument tested needs to be of the class `asConstInt`.

6.6 Combining Static and Dynamic Benefits

Now that we have the frequency information, the estimate of run-time values and the method summary, it is possible to aggregate these information into a single notion of inlining "benefit". First, as mentioned in Chapter 4 and Chapter 6, the method summaries are used as constraints and the abstract values are used to see whether those constraints are satisfied at each call site. To generate a value out of the method summaries, each row of the method summary is augmented with a weight. The weight corresponds to the benefit value associated with a row's inlining benefit. It is the job of the compiler developer to create a meaningful weight, but it must be a positive integer value.

For example, the method summary in Table 6.4 can be extended as shown in Table 6.7. A new column, weight, shows variable w that refers to how beneficial this optimization is. If the constraint related to the argument is

Table 6.7: Example method summary to illustrate how argument estimates and argument constraints interact

Potential Opt	**Weight**	**Arg 0**	**Arg 1**	**Arg 2**
Instance Of	w_0	`subset(BaseClass)`		
Branch Folding	w_1		[`INT_MIN`, -1]	
Branch Folding	w_2		[0, `INT_MAX`]	
Null check	w_3			`isNull`
Null check	w_4			`isNonNull`

	Top of stack
Argument 2	$\top$
Argument 1	$[1, 100]$
Argument 0	`DerivedClass`
	Bottom of stack

Figure 6.8: Abstract stack containing abstract argument estimates.

satisfied then the weight w_c is added to an aggregate. The bytecode location column has been omitted to save space. The estimate to the argument values obtained through the abstract interpretation are found in Table **??**.

With this information, it is possible to aggregate all satisfied constraints into a single value. The next step is to iterate over every row in the method summary, check to see if the constraint is satisfied, and if the constraint is satisfied aggregate the weight values into a single value. For example, argument at position 0 is a derived class from `BaseClass` which is stored to be used as an argument on the constraint. Argument at position 1 has an estimate of an integer value from 0 to 100. The second entry of the method summary is not satisfied, but the third one is. Finally, the estimate of argument at position 2 is $\top$. Because $\top$ is neither `isNull` nor `isNonNull`, then the last entries of the method summary are not satisfied. The weights w_0 and w_2 are added into what we consider the *static benefit of inlining*.

The next step is to combine the static benefits of inlining with the dynamic benefits of inlining which is called *callRatio* in Section 4.2. There are an infinite

number of functions of the form:

$$f : \textit{static benefits} \times \textit{call ratio} \rightarrow \mathbb{Z}^{+}$$

Different functions represent different design decisions and may give different weight to the inlining candidates. We have chosen to multiply the *callRatio* by the *static benefit of inlining*.

$$f(\textit{static benefits}, \textit{call ratio}) = \textit{static benefits} \times \textit{call ratio}$$

This definition for this function scales the call ratio by the static benefits.

The procedure outlined in by Craik et al. takes as an input an IDT with each node annotated with a cost and a benefit [16]. A procedure to determine the cost and the benefit has been covered in this thesis. The procedure outlined in by Craik et al. can then be used with the weights outlined in this thesis. Furthermore, design decisions have been explicitly pointed out to allow exploration of the design space of this solution.

6.6.1 Tuning

An interesting property of grading inlining transformations using method summaries is the ability to tune inlining decisions based on user parameters. For example, if a user is interested only a particular optimization, then setting the weights of all optimizations to zero with the exception of the optimization of interest, allows the inliner to assign a higher probability of inlining those call-sites which will allow that optimization to happen. While this is an interesting property, it has not been implemented thoroughly. Future work includes adding support for run time selection of weights to optimization based on optional parameters.

6.7 Summary

In summary, we have developed a systematic way to estimate if some optimizations will take place during compilation. Most importantly, these optimizations are conditional upon methods being inlined. That is, these optimizations are

examples of the *indirect benefits* of inlining. These optimizations are encoded in a data structure that is human readable. Method summaries help compiler developers understand why a method was inlined.

The optimizations considered in the previous sections are only a proof of concept. The set of optimizations that are considered by the method summaries can be expanded in future work. In the simple examples shown previously, there was only one type of optimization encoded in each method summary, however, in the general case method summaries contain multiple rows denoting multiple optimizations. A method summary may encode different types of optimizations that are unlocked by different arguments. Constraints on multiple arguments would be equivalent to having multiple non-empty columns in the same row of the method summary.

Limitations in method summaries exist. For example, at the moment only a single argument is constrained (i.e., only a single cell in a method summary entry is a constraint). Figure 6.9 shows bytecode that contains a branch (`5: ifge` which depends on the target address of `1: ifle`. Our current algorithm will generate the method summary found in Table 6.8. If an argument estimate satisfies one of the constraints for `5 : ifge`, then that branch would be folded. However, it might be of interest to calculate when both constraints are satisfied in the way shown by Table 6.9 which is currently beyond the capabilities of the current implementation. E.g., our current algorithm can only generate method summary entries with a single constraint (see rows in Table 6.8). Generating method summary entries with multiple constraints (see third row in Table 6.9) per row would allow the current implementation to determine whether conditional statements that test multiple values can be folded away, as opposed to conditional statements that test a single value.

```
1  public static int condition(int, int);
2    Code:
3       0: iload_0
4       1: ifle         12
5       4: iload_1
6       5: ifge         12
7       8: iconst_1
8       9: goto         13
9      12: iconst_0
10     13: istore_2
11     14: iload_2
12     15: ifeq         20
13     18: iconst_1
14     19: ireturn
15     20: iconst_0
16     21: ireturn
```

Figure 6.9: Example of branch being conditional on multiple arguments

Table 6.8: Method summary for Figure 6.9

Potential Optimization	**Bytecode**	**Argument 0**	**Argument 1**
Branch Folding	`1 : ifle`	[`INT_MIN`, 0]	
Branch Folding	`1 : ifle`	[1,`INT_MAX`]	
Branch Folding	`5 : ifge`		[`INT_MIN`, 0]
Branch Folding	`5 : ifge`		[1,`INT_MAX`]

Table 6.9: Method summary for Figure 6.9 after future work

Potential Optimization	**Bytecode**	**Argument 0**	**Argument 1**
Branch Folding	`1 : ifle`	[`INT_MIN`, 0]	
Branch Folding	`5 : ifge`		[1,`INT_MAX`]
Branch Folding	`1-5`	[1,`INT_MAX`]	[`INT_MIN`, 0]

Chapter 7

Evaluation

This performance evaluation contains two comparisons: one against the OpenJ9 JVM to compare the performance of the benefit inliner against an industry-grade inliner, and another against a call-ratio inliner, which uses only frequency information. The inlining strategies used in the OpenJ9 JVM are publicly available as part of the OpenJ9 JVM source code. The OpenJ9 JVM includes two inlining strategies: `TR_DumbInliner` is used only during `cold` compilation levels, and `TR_MultipleCallTargetInliner` is used for the compilation levels `warm` and above.

The call-ratio inliner is a modified version of the benefit inliner where only the call ratio is used to determine what to inline. This comparison allows us to measure differences in the inlining plans when taking into account the predicted optimizations.

7.1 Experimental Setup

This performance evaluation uses a subset of the DaCapo 9.12 Bach [11] benchmarking suite for the evaluation. The DaCapo benchmarking suite is popular, relevant, and has a diverse set of workloads. The following benchmarks were excluded from the evaluation: TOMCAT, TRADEBEANS, TRADESOAP. The evaluation is conducted on a machine equipped with an Intel Xeon Platinum 8180 processor [28]. The machine is configured such that 28 cores are active. The machine has an L1 data cache and an L1 instruction cache of 32K each. The L2 cache is 1,024K and L3 cache is 39,424K, and RAM is 1TB. IBM

```
1   if (isScorching(comp()))  _callerWeightLimit = std::max(1500,
       size * 2);
2   else if (isHot(comp()))   _callerWeightLimit = std::max(1500,
       size + (size >> 2));
3   else if (size < 125)      _callerWeightLimit = 250;
4   else if (size < 700)      _callerWeightLimit = std::max(700,
       size + (size >> 2));
5   else _callerWeightLimit = size + (size >> 3);
```

Figure 7.1: Amount of budget allocated for inlining as a function of the method's size and compilation level is given by the variable `_callerWeightLimit`. [21][Inliner.cpp, Line 311]

granted access to this machine with the server configured in such a way that the ports used by the excluded benchmarks were closed.

Red Hat Enterprise Linux Server release 7.4 (Maipo) is installed as the machine's operating system (OS) and is running kernel version 3.10.0-693. The machine is running an OpenJ9 JVM running Java version 1.8.0_171 [1]. The OpenJ9 JVM has been modified to include the prototype implementations of the benefit inliner and the call-ratio inliner.

We have taken care to match the inlining budgets used in the benefit-driven and call-ratio inliner to the budgets used in the OpenJ9 JVM. However, the inlining strategies used in the OpenJ9 JVM conflate the notion of benefit and size into a weighted size. Furthermore, the OpenJ9 JVM inlining strategies include a heuristic for ignoring the budget for certain call-site/callee pairs. As a result, while we have attempted to match the budgets used by in the different inlining strategies, there are still some differences in the inlining budgets. Changing the OpenJ9 JVM inlining strategy to avoid conflating the notion of benefit and size would not be advisable since that is part of our contribution. Changing the OpenJ9 JVM inlining strategy to avoid ignoring the budget for certain call-site callee pairs would make the comparison unfair. As such, the inlining strategies used in the OpenJ9 JVM remain unchanged.

The budget is given by the following function available in the

[1]The commit hashes are OpenJ9: 615f0cc, OMR: 7a158d9. The JCL version is: 20180604_01.

Table 7.1: Warm up iterations and repetitions for each benchmark

Benchmark	Warm up iterations
AVRORA	60
BATIK	100
ECLIPSE	10
FOP	1,000
H2	100
LUINDEX	600
LUSEARCH	100
PMD	100
SUNFLOW	100
XALAN	400

7.2 Following Best Practices

To avoid measuring the effects of stop-the-world garbage collection we set the JVM heap to be 1 GB. To avoid non-determinism introduced by Non-Uniform Memory Access (NUMA), all 28 cores have been selected to run under a single NUMA node. NUMA allows processors in multiprocessor architectures to access their own local memory faster than non-local memory. Because threads may be scheduled in a non-deterministic way, NUMA may introduce some noise. As such, we would like to reduce the noise introduced by NUMA. To achieve this we disable multithreading on multiple processors. There were 28 threads each assigned to a single microprocessor ensuring that there was uniform memory access.

All benchmarks run until the JVM ends warming up. Table 7.1 shows the number of warm up iterations for each benchmark. After warm up, the benchmark runs one more time and the run time reported by this last run is recorded. The number of iterations a benchmark needs to run to be considered warmed up was obtained by measuring the compiler activity. The JVM is considered to finish the warming up stage when the JVM runs at least 3 iterations of the benchmark without any compilation requests or no new compilation requests were issued in the last thirty seconds of execution.

The benchmarks were executed in an isolated environment. To account

for variations in execution time introduced by background processes, each measurement is repeated 10 times. The value reported is the arithmetic mean of ten runs. The recorded points for each run is modelled as a gaussian distribution and the standard deviation is reported in the graphs. A single execution batch consists in running each one of the benchmarks once in a given order. The order of execution of the benchmarks is randomized from one batch to the next. Benchmarks are run back to back (i.e., once a benchmark finishes executing, the next one is scheduled to run).

Source of variations also include the use of sampling profiling to determine the frequency of execution of methods and trigger compilation requests and profiling information used in the inlining decisions. Previous inlining decisions may also affect the profiling information because once a method has been inlined, it no longer gets profiled by the IProfiler framework. Variations may be introduced by the thread scheduler because the OpenJ9 JVM uses asynchronous compilation. These sources of variation affect the inliners by potentially changing inlining decisions.

7.3 Measurements

7.3.1 Run Time

The DaCapo benchmarks report run time as an output during each iteration of the benchmark. We run the benchmarks multiple times using the command line options available in the DaCapo benchmarking suite. These multiple runs ensure that the virtual machine has finished warming up. The reported time is normalized against the baseline inliner.

The geometric mean across all benchmarks for the baseline inliner is 1 ± 0.009. The geometric mean across all benchmarks for the benefit inliner is 1.045 ± 0.006. This means that the benefit inliner is 4% slower than the baseline inliner.

For the majority of the benchmarks, there is little impact on the run time compared with the baseline inliner. However, for the benchmark FOP, the run time increased by almost 20% when running the benefit inliner and the call-ratio inliner when compared to the baseline inliner. Similarly, benchmarks

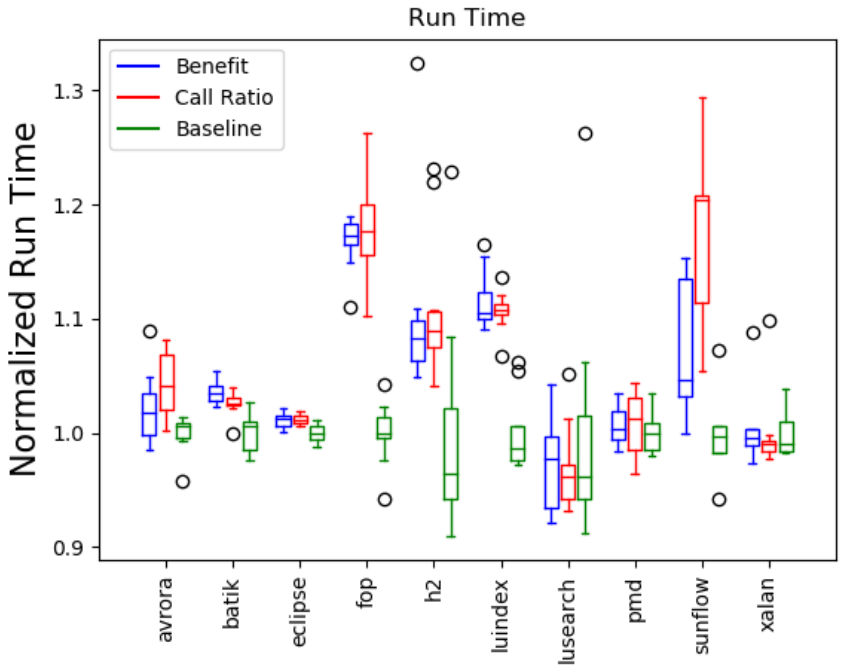


Figure 7.2: Normalized run time: average of 10 runs for baseline, call ratio, and benefit inliner

H2 and LUINDEX also had run time increased by more than 10%.

The geometric mean across all benchmarks for the call-ratio inliner is 1.056 ± 0.007. The difference in run time between the call-ratio inliner and the benefit inliner is negligible.

7.3.2 Compilation Time

OpenJ9 provides infrastructure to obtain the compilation time spent by each compilation thread. The total compilation time is obtained by adding the compilation time reported by each compilation thread. Figure 7.3 shows the average compilation time of each benchmark with their respective standard deviation. The compilation time is normalized to the average of the baseline.

The geometric mean across all benchmarks for the baseline inliner is 1 ± 0.017. The geometric mean across all benchmarks for the call-ratio inliner is 1.119 ± 0.026. The geometric mean across all benchmarks for the benefit inliner is 1.102 ± 0.024. We include the geometric mean for the baseline inliner to show the difference in the standard deviation between the different inliners.

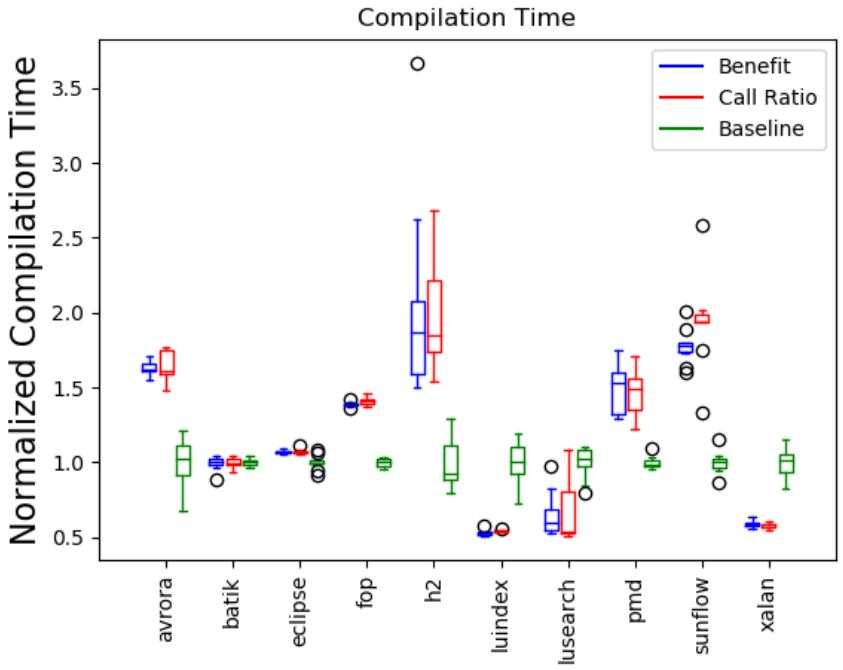


Figure 7.3: Normalized compilation time: average of 10 runs for baseline inliner, call-ratio inliner, and benefit inliner

Figure 7.3 shows a large variation when comparing the benefit inliner and the call-ratio inliner against the baseline inliner. Some of this increase in time is attributed to the time spent analyzing code. However, as mentioned in Section 7.1, there are some fundamental differences in the notion of inlining budget between the benefit inliner and the baseline inliner. The differences are that the budget in the OpenJ9 JVM inliners conflates the size of the method to be inlined with the frequency of execution of that method. Thus, the budget in the OpenJ9 JVM corresponds to the notion of the number of bytecodes in a method and how hot the method is. For example, when determining which methods to inline, the inliner determines if the frequency of invocation of a method, and if it surpasses a threshold, then the inliner multiplies the size of the bytecode by a value less than 1 to make this method more likely to be inlined.

7.3.3 Difference in Factors Influencing Inlining

In Chapter 2 introduced the inlining strategies already available in OpenJ9. The heuristic properties used by the OpenJ9 inliner are spread throughout the OpenJ9 repository. The OpenJ9 inlining strategies adjust the weight of inlining candidates considering the following properties:

1. heavily polymorphic interfaces
2. *invokedynamic* instructions
3. hotness
4. bytecode size
5. polymorphic callee sizes
6. polymorphic root sizes
7. frequency
8. constant arguments
9. number of callers to a method
10. inline depth
11. number of inlined call sites
12. call site is in a loop
13. number of nodes in the intermediate language

Some of these properties are obtained after generating the intermediate language in OpenJ9. This means that time has been spent translating from bytecode to intermediate language. Furthermore, methods transformed to intermediate language in the inlining pass will need to be re-compiled to intermediate language due to internal implementation details. Because the benefit inliner works at the byte code level, we are allowed to save time needed to compile to Trees (the intermediate language of OpenJ9). However, due to the time spent in the analysis, we still have a slow down of 10%.

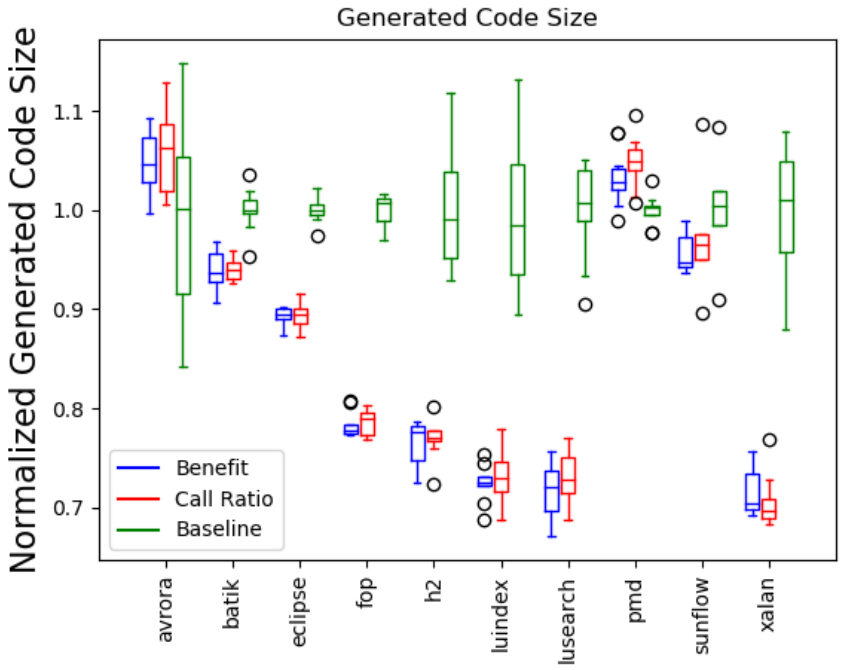


Figure 7.4: Normalized generated code

7.3.4 Generated Code Size

Compilation time is usually proportional to the amount of generated code size. This is because many of the different analyses used during compilation have super-linear time complexity on the size of the code [10]. As a result, in order to estimate the compilation time, we must also look at the amount of code compiled.

The size of generated code is obtained through the verbose option from the same 10 runs of each benchmark. The verbose option outputs the start and end addresses for each compiled method body. The size of generated code for each compilation request is given by the difference between the end and the start addresses. The total size of generated code for each benchmark run is obtained by adding the generated code for each compilation issued during that benchmark run. Figure 7.4 shows the average amount of generated code during the execution of 10 runs of each benchmark.

There is a moderate correlation (coefficient 0.522) between the amount of generated code and the amount of compilation time for the benefit inliner.

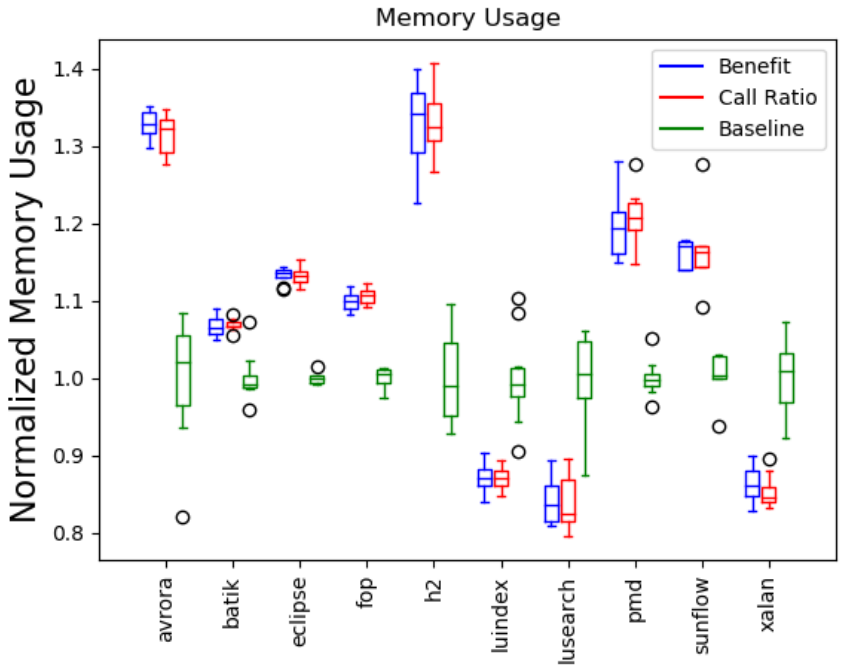


Figure 7.5: Memory usage

This means that more code generated implies a higher compilation time for the benefit inliner and the call-ratio inliner.

The geometric mean of the generated code size for the call-ratio inliner across all benchmarks is 0.845 ± 0.005. The geometric mean of the generated code size for the benefit inliner across all benchmarks is 0.848 ± 0.004. On average, using the benefit inliner produces 16% less code and increases run time by 4%. This variability is high, but the amount of generated code is usually equal or less than the amount of code generated by the baseline inliner. The decrease in generated code size is attributed to the baseline inliner's conflation of a method size with the method frequency of invocation.

7.3.5 Memory Usage

Figure 7.5 shows the amount of memory used during compilation. This value was obtained from the logging infrastructure available in the JIT used in OpenJ9 JVM.

The geometric mean across all benchmarks for the baseline inliner is $1 \pm$

0.017. The geometric mean across all benchmarks for the benefit inliner is 1.074 ± 0.004. The geometric mean across all benchmarks for the call-ratio inliner is 1.074 ± 0.005.

Figure 7.5 shows that the impact of the analysis on the memory consumption of the compiler is 7%. This difference is small taking into account the greater amount of search space that is explored. While the benefit inliner and the call-ratio inliner performs a more complex analysis than the inliners in the OpenJ9 JVM, the memory consumption increases by an average of only 7%.

7.4 Case Studies

Section 7.3 compares the overall performance of the benefit inliner against a state of the art inlining strategies. However, these measurements indicate that the differences between the call-ratio inliner and the benefit inliner are not statistically significant. The Wilcoxon signed-rank test indicates that there are no differences in compile time, run time, memory usage and generated code size between the benefit inliner and the call-ratio inliner ($Z = 2133, p = 0.17$ $Z = 1639, p = 0.18$, $Z = 2046, p = 0.38$, $Z = 1976, p = 0.33$ respectively). Cliff's delta for these groups are between -0.02 and 0.05 which corresponds to a negligible difference [36] (as cited in in [45]) Why is this the case? Are the inlining plans between the call-ratio inliner and the benefit inliner the same?

Answering this question is difficult, because inlining plans for the same compilation unit across multiple runs may differ even using the same inliner. Ideally, we would like to have the same inlining plan across different runs. However, due to non-determinism discussed in Section 7.2, inlining plans can be affected in the following ways:

1. Considering the same nodes in the IDT but each node in the IDT may have different values for the block frequencies.

2. Considering different nodes in the IDT. While the inliner normally considers all nodes for inclusion in the IDT, there are two exceptions: (i) When considering virtual and interface invocations, only the main target

of these methods is added to the IDT and (ii) cold methods are excluded from the IDT.

3. The VM is in control of which methods it issues for compilation/re-compilation which is also a non-deterministic process.

As a result of these sources of uncertainty, it is impossible to obtain fully reproducible inlining plans for all compilation units.

To find out how different the inlining plans generated between the benefit inliner and the call-ratio inliner, one must look at the following subset of compilation units:

1. Concentrate on inlining plans in which the abstract interpreter found at least one optimization.

2. Look at the top 100 hot methods.

3. Look for compilation units in the benefit inliner whose inlining plans are different for the majority of runs when compared to the call-ratio inliner.

4. Look for the intersection of compilation units found in the benefit inliner and the call-ratio inliner.

There is little difference between the benefit inliner and the call-ratio inliner when looking at that subset of compilation units. After manual inspection, the most prominent examples are outlined here as use cases. These methods are the ones where the computed benefit by the analysis lead to better inlining decisions compared to just considering the call ratio.

7.4.1 `arrayAtPut()`

The DaCapo benchmark ECLIPSE has the method `arrayAtPut(int, boolean)`. The source of this method can be found in the `CodeStream` class in the `eclipse` repo [20] and is also shown in Figure 7.6. In the compilation requests for `ArrayInitializer.generateCode()`, `arrayAtPut(int, boolean)` has a frequency value similar to other methods. However, due to its relatively large amount of bytecode footprint, the call-ratio inliner decided against

inlining it. The abstract interpreter is capable of determining that the argument `valueRequired` will always be `false` when compiling `ArrayInitializer.generateCode()` and that it leads to folding 8 branches. As a result, the benefit inliner consistently inlines this method instead of several other smaller methods. This is a prime example of the type of methods that the analysis in the benefit inliner discovers.

7.4.2 `renderInlineArea()`

The method `renderInlineArea(InlineArea inlineArea)` can be found in the `AbstractRenderer` class in the FOP repo and is shown in Figure 7.7. The benefit inliner was able to determine that the instruction `instanceof` was being used heavily in this method and that the class of `inlineArea` can be known at analysis time. As a result, `if` statements in this method can be folded away.

7.4.3 `regionMatches()`

In LUSEARCH, when compiling root method `SegmentInfos.read(Directory d,String s)` both inliners consider inlining `generationFromSegmentsFileName(String fileName)`. This method calls `String.startsWith(String prefix, int start)` (source available in Figure 7.8) with a `static final` string obtained from the field `IndexFileNames.SEGMENTS`. There are other literal values in the call chain. For example, the arguments `thisStart` and `start` to the function `String.regionMatches(int, String, int, int)` in line 18 come from literal values from ancestors in the call graph. The current implementation of the abstract interpreter does not model field accesses. Therefore, it cannot determine the value of the `string` argument, but it can determine that the arguments `thisStart` and `start` are constants and their respective values. As such, the branches in Lines 20 and 21 are simplified to eliminate the inequality comparison for `thisStart` and `start`.

The call-ratio inliner considers inlining `regionMatches()` but the call ratio alone is not beneficial enough to inline `regionMatches()` in any of the runs. The static benefits found by the benefit inliner provide sufficient weight to

```
1  public void arrayAtPut(int elementTypeID, boolean valueRequired) {
2    switch (elementTypeID) {
3      case TypeIds.T_int :
4        if (valueRequired)
5          dup_x2();
6        iastore();
7        break;
8      case TypeIds.T_byte :
9      case TypeIds.T_boolean :
10       if (valueRequired)
11         dup_x2();
12       bastore();
13       break;
14     case TypeIds.T_short :
15       if (valueRequired)
16         dup_x2();
17       sastore();
18       break;
19     case TypeIds.T_char :
20       if (valueRequired)
21         dup_x2();
22       castore();
23       break;
24     case TypeIds.T_long :
25       if (valueRequired)
26         dup2_x2();
27       lastore();
28       break;
29     case TypeIds.T_float :
30       if (valueRequired)
31         dup_x2();
32       fastore();
33       break;
34     case TypeIds.T_double :
35       if (valueRequired)
36         dup2_x2();
37       dastore();
38       break;
39     default :
40       if (valueRequired)
41         dup_x2();
42       aastore();
43   }
44 }
```

Figure 7.6: Java source for inlining candidate `arrayAtPut()`.

```
protected void renderInlineArea(InlineArea inlineArea) {
        List<ChangeBar> changeBarList = inlineArea.
            getChangeBarList();

        if (changeBarList != null && !changeBarList.isEmpty()) {
            drawChangeBars(inlineArea, changeBarList);
        }
        if (inlineArea instanceof TextArea) {
            renderText((TextArea) inlineArea);
        //} else if (inlineArea instanceof Character) {
            //renderCharacter((Character) inlineArea);
        } else if (inlineArea instanceof WordArea) {
            renderWord((WordArea) inlineArea);
        } else if (inlineArea instanceof SpaceArea) {
            renderSpace((SpaceArea) inlineArea);
        } else if (inlineArea instanceof InlineBlock) {
            renderInlineBlock((InlineBlock) inlineArea);
        } else if (inlineArea instanceof InlineParent) {
            renderInlineParent((InlineParent) inlineArea);
        } else if (inlineArea instanceof InlineBlockParent) {
            renderInlineBlockParent((InlineBlockParent) inlineArea)
                ;
        } else if (inlineArea instanceof Space) {
            renderInlineSpace((Space) inlineArea);
        } else if (inlineArea instanceof InlineViewport) {
            renderInlineViewport((InlineViewport) inlineArea);
        } else if (inlineArea instanceof Leader) {
            renderLeader((Leader) inlineArea);
        }
}
```

Figure 7.7: Java source for inlining candidate `renderInlineArea()`.

the nodes in the algorithm such that `regionMatches()` is inlined in all the runs. Similar examples can be found by the benefit inliner across different benchmarks where only the benefit inliner chooses to inline `regionMatches()`.

7.4.4 `loadClassHelper()`

The benchmark PMD uses the function `loadClass(final String className, boolean resolveClass)` from the Java standard library. The source is available in Figure 7.9. The specific implementation found in the OpenJ9 repo [22] contains the helper function `loadClassHelper()` that is consistently inlined in the benefit inliner but not consistently enough in the call-ratio inliner. The boolean value `delegateToParent` can be found to be `true` by the benefit inliner every time the `loadClass(final String, boolean)` and `loadClassHelper(String, boolean, boolean)` are inlining candidates.

7.4.5 `StringBuilder()`

In SUNFLOW and AVRORA and other benchmarks, there are several uses of `StringBuilder`. The value `INITIAL_SIZE` can be determined at compilation time and can be used to fold away the branch in Line 4 in Figure 7.10.

7.4.6 `getZero()`

In the PMD benchmark, we found that the function `getZero(Locale l)` can determine that `Locale l` is not `null`. The `new` instruction found in Line 2 of Figure 7.11 when passing the second argument maps to the non `null` abstract value for the class `StringBuilder`. Lines 7 and 8 have no effect on the abstract interpreter except for manipulation of the stack. However, the call in Line 9 passes a non `null` abstract value for the object allocated in Line 2. The method summary for the `getZero(Locale l)` function includes a `null` check on argument `Locale l`. The constraint from the method summary is satisfied when checked against the abstract argument generated in Line 2.

```
1 public static long generationFromSegmentsFileName(String fileName)
      {
2  if (fileName.equals(IndexFileNames.SEGMENTS)) {
3   return 0;
4  } else if (fileName.startsWith(IndexFileNames.SEGMENTS)) {
5   return Long.parseLong(fileName.substring(1+IndexFileNames.
       SEGMENTS.length()),
6      Character.MAX_RADIX);
7  } else {
8   throw new IllegalArgumentException("fileName \"" + fileName + "
       \" is not a segments file");
9  }
10 }
11
12 public boolean startsWith(String prefix) {
13  return startsWith(prefix, 0);
14 }
15 public boolean startsWith(String prefix, int start) {
16  return regionMatches(start, prefix, 0, prefix.count);
17 }
18 public boolean regionMatches(int thisStart, String string, int
     start, int length) {
19  string.getClass(); // implict null check
20  if (start < 0 || string.count - start < length) return false;
21  if (thisStart < 0 || count - thisStart < length) return false;
22  if (length <= 0) return true;
23  int o1 = offset + thisStart, o2 = string.offset + start;
24  int end = length - 1;
25  char[] source = value;
26  char[] target = string.value;
27  target.getClass(); // implicit null check
28  source.getClass(); // implicit null check
29  // fast path check - strings are much more likely to be different
       at the end
30  if (source[o1 + end] != target[o2 + end]) return false;
31  for (int i = 0; i < end; ++i) {
32   if (source[o1 + i] != target[o2 + i])
33    return false;
34  }
35  return true;
36 }
```

Figure 7.8: Java source for inlining candidate `regionMatches()`.

```
protected Class<?> loadClass(final String className, boolean
    resolveClass) throws ClassNotFoundException {
  return loadClassHelper(className, resolveClass, true)
}
Class<?> loadClassHelper(final String className, boolean
    resolveClass, boolean delegateToParent
  ) throws ClassNotFoundException {
  Object lock = isParallelCapable ? getClassLoadingLock(className)
      : this;
  synchronized (lock) {
    // Ask the VM to look in its cache.
    Class<?> loadedClass = findLoadedClass(className);
    // search in parent if not found
    if (loadedClass == null) {
      if (delegateToParent) {
        try {
          if (parent == null) {
            /*[PR 95894]*/
            if (isDelegatingCL) {
              loadedClass = bootstrapClassLoader.findLoadedClass(
                  className);
            }
            if (loadedClass == null) {
              loadedClass = bootstrapClassLoader.loadClass(
                  className);
            }
          } else {
            if (isDelegatingCL) {
              loadedClass = parent.findLoadedClass(className);
            }
            if (loadedClass == null) {
              loadedClass = parent.loadClass(className,
                  resolveClass);
            }
          }
        } catch (ClassNotFoundException e) {
          // don't do anything. Catching this exception is the
              normal protocol for
          // parent classloaders telling use they couldn't find a
              class.
        }
      }
      if (loadedClass == null) {
          loadedClass = findClass(className);
      }
    }
    if (resolveClass) resolveClass(loadedClass);
    return loadedClass;
  }
}
```


Figure 7.9: Java source for inlining candidate `loadClassHelper()`.

```
1 public StringBuilder() {
2   this(INITIAL_SIZE);
3 }
4
5 public StringBuilder(int capacity) {
6   if (capacity < 0) {
7     throw new NegativeArraySizeException();
8   }
9   int arraySize = capacity;
10
11  if (String.enableCompression) {
12    if (capacity == Integer.MAX_VALUE) {
13      arraySize = (capacity / 2) + 1;
14    } else {
15      arraySize = (capacity + 1) / 2;
16    }
17  }
18  value = new char[arraySize];
19
20  this.capacity = capacity;
21 }
```

Figure 7.10: Java source for inlining candidate `StringBuilder()`.

```
1     public Formatter() {
2         this(Locale.getDefault(Locale.Category.FORMAT), new
              StringBuilder());
3     }
4
5     private Formatter(Locale l, Appendable a) {
6         this.a = a;
7         this.l = l;
8         this.zero = getZero(l);
9     }
10
11    private static char getZero(Locale l) {
12        if ((l != null) && !l.equals(Locale.US)) {
13            DecimalFormatSymbols dfs = DecimalFormatSymbols.
                  getInstance(l);
14            return dfs.getZeroDigit();
15        } else {
16            return '0';
17        }
18    }
```

Figure 7.11: Java source for inlining candidate `getZero()`.

Chapter 8

Conclusion

This thesis shows a systematic way of assigning values to inlining candidates based on their frequency of execution and the optimizations that will take place after inlining. While data-flow analyses are not normally used in the JIT context due to the compiler's time constraints, our benefit inliner performs reasonably well compared to the greedy inlining strategy. However, there are low number of different inlining plans when comparing the benefit inliner to the call-ratio inliner.

Throughout the course of this thesis, abstract interpretation, a theory of data-flow analyses, has been used to design a fast static analysis with the purpose of estimating the benefit of inlining decisions. The benefit inliner combines the direct and indirect benefits of inlining into a single benefit value. Additionally, we use a method summary to encode constraints that improve the likelihood that a method will be inlined. Method summaries are a good way of encoding the benefits of inlining, because they are easily understandable.

This thesis has shown a lower bound on the quantity of inlining candidates that are benefitted by this analysis. While the amount of constraints that were satisfied were relatively low, the static analysis performed in the benefit inliner lacks precision. This thesis has outlined future work to improve the precision and usability of the benefit inliner and increase the quantity of satisfied constraints.

8.1 Future Work

The inlining strategy described in this work is just a prototype. Future work can be categorized the following different areas:

1. validate design decisions through experimentation
2. add support for more optimizations
3. increase analysis' precision
4. optimize current implementation
5. extend method summaries

Was it a good decision to use the same inlining budget on the call ratio and the benefit inliner as the default inliners on OpenJ9? Sweeping possible budget values and finding out the impact of varying the budget on run time and compilation time can provide evidence to decide on better budgets for inlining. Another design decision could be interesting to explore is being more selective on the call-site/method pairs that are added to the IDT. Depending on the budget, the IDT may be bigger than 1,500 nodes. While the IDT considers all nodes to find which ones should be inlined, maybe heuristics could help to narrow down the search space before running inlining packing algorithm. It could also be possible to limit the growth of the IDT by separating the budget into two: one budget allows for exploring (which will limit the size and depth of the IDT) while the other one allows for inlining within the explored search space.

The current number of optimizations is quite limited and may provide limited benefits. We are currently exploring the possibility of adding support for escape analysis. Finding opportunities to allocate elements on the stack as opposed to on the heap may reduce run time. Other optimizations worth considering is looking at the return values of functions. In some cases it might be possible that the return type of a function is a subtype of the function's type signature. In these cases, it might be possible to devirtualize calls when the return type is the target of the call.

The benefit inliner strategy discards the state in the case of back-edges. This limits the precision of the inliner. We are currently exploring the possibility of doing two passes on the control-flow graph to allow for data flow facts to be propagated by back-edges. It might be worthwhile to consider an experiment where the analysis is performed until the fixed point is reach to find the upper bound on post-inlining transformations discovered on the benchmarks.

Finally, the benefit inliner strategy is in the process of being open sourced. We are communicating with the OpenJ9 team to make sure that our prototype is optimized. Futhermore, method summaries are limited in their capacity to represent constraints. We will be working on extending method summaries to allow more expressive constraints.

References

[1] E. K. (https://stackoverflow.com/users/223429/eugene-kuleshov), *How can the jvm verify there's no potential operand stack overflow when loading a class?* Stack Overflow, https://stackoverflow.com/a/10541774. eprint: https://stackoverflow.com/a/10541774. [Online]. Available: https://stackoverflow.com/a/10541774. 6

[2] R. I. (https://stackoverflow.com/users/225757/roland-illig), *Java: How to check for null pointers efficiently*, Stack Overflow, https://stackoverflow.com/q/4795455. eprint: https://stackoverflow.com/q/4795455. [Online]. Available: https://stackoverflow.com/q/4795455. 46

[3] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp, "The jikes research virtual machine project: Building an open-source research community," *IBM Systems Journal*, vol. 44, no. 2, pp. 399–417, 2005, ISSN: 0018-8670. DOI: 10.1147/sj.442.0399. [Online]. Available: https://doi.org/10.1147/sj.442.0399. 4

[4] A. W. Appel, *Compiling with continuations*. Cambridge University Press, 2006. 23

[5] M. Arnold, S. Fink, V. Sarkar, and P. F. Sweeney, "A comparative study of static and profile-based heuristics for inlining," *SIGPLAN Not.*, vol. 35, no. 7, pp. 52–64, Jan. 2000, ISSN: 0362-1340. DOI: 10.1145/351403.351416. [Online]. Available: http://doi.acm.org/10.1145/351403.351416. 1, 21–23

[6] M. Arnold and B. G. Ryder, "A framework for reducing the cost of instrumented code," *SIGPLAN Not.*, vol. 36, no. 5, pp. 168–179, May 2001, ISSN: 0362-1340. DOI: 10.1145/381694.378832. [Online]. Available: http://doi.acm.org/10.1145/381694.378832. 9

[7] J. Aycock, "A brief history of just-in-time," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 97–113, Jun. 2003, ISSN: 0360-0300. DOI: 10.1145/857076.857077. [Online]. Available: http://doi.acm.org/10.1145/857076.857077. 5

[8] Azul, *Zing jvm*, https://www.azul.com/products/zing/, 2019. 4

[9] beehive-lab, *Maxine vm*, https://github.com/beehive-lab/Maxine-VM, 2019. 4

[10] P. Berube, "Methodologies for many-input feedback-directed optimization," AAINR89287, PhD thesis, Edmonton, Alta., Canada, 2012, ISBN: 978-0-494-89287-9. DOI: 10.7939/R3DW8K. [Online]. Available: https://doi.org/10.7939/R3DW8K. 1, 21, 23, 28, 62

[11] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The dacapo benchmarks: Java benchmarking development and analysis," *SIGPLAN Not.*, vol. 41, no. 10, pp. 169–190, Oct. 2006, ISSN: 0362-1340. DOI: 10.1145/1167515.1167488. [Online]. Available: http://doi.acm.org/10.1145/1167515.1167488. 55

[12] E. Bodden, "The secret sauce in efficient and precise static analysis: The beauty of distributive, summary-based static analyses (and how to master them)," in *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, ser. ISSTA '18, Amsterdam, Netherlands: ACM, 2018, pp. 85–93, ISBN: 978-1-4503-5939-9. DOI: 10.1145/3236454.3236500. [Online]. Available: http://doi.acm.org/10.1145/3236454.3236500. 24

[13] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W.-m. W. Hwu, "Profile-guided automatic inline expansion for c programs," *Softw. Pract. Exper.*, vol. 22, no. 5, pp. 349–369, May 1992, ISSN: 0038-0644. DOI: 10.1002/spe.4380220502. [Online]. Available: http://dx.doi.org/10.1002/spe.4380220502. 21

[14] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '77, Los Angeles, California: ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973. [Online]. Available: http://doi.acm.org/10.1145/512950.512973. 10, 13

[15] A. Craik, https://github.com/eclipse/openj9/issues/199#issuecomment-334290558, [Online; accessed March-08-2019], 2017. 9

[16] A. J. Craik, R. E. Craik, and P. R. Doyle, *Expanding inline function calls in nested inlining scenarios*, US Patent App. 15/245,241, Jul. 2017. ii, 17, 26, 28, 52

[17] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko, "Compiling java just in time," *IEEE Micro*, vol. 17, no. 3, pp. 36–43, May 1997, ISSN: 0272-1732. DOI: 10.1109/40.591653. [Online]. Available: doi.org/10.1109/40.591653. 4, 5

[18] J. Dean and C. Chambers, "Towards better inlining decisions using inlining trials," *SIGPLAN Lisp Pointers*, vol. VII, no. 3, pp. 273–282, Jul. 1994, ISSN: 1045-3563. DOI: 10.1145/182590.182489. [Online]. Available: http://doi.acm.org/10.1145/182590.182489. 2, 22–24

[19] R. Dubisch, "Lattices to logic," 1964. 12

[20] Eclipse, *Eclipse jdt core*, https://github.com/eclipse/eclipse.jdt.core, 2019. 65

[21] ——, *Eclipse omr*, https://github.com/eclipse/omr, 2019. 8, 56

[22] ——, *Eclipse openj9*, https://github.com/eclipse/openj9, 2019. 4, 7, 69

[23] I. Gartley, M. Pirvu, V. Sundaresan, and N. Grcevski, "Experiences in designing a robust and scalable interpreter profiling framework," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Feb. 2013, pp. 1–10. DOI: 10.1109/CGO.2013.6494981. 9

[24] M. Gaudet and M. Stoodley, "Rebuilding an airliner in flight: A retrospective on refactoring ibm testarossa production compiler for eclipse omr," in *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages*, ser. VMIL 2016, Amsterdam, Netherlands: ACM, 2016, pp. 24–27, ISBN: 978-1-4503-4645-0. DOI: 10.1145/2998415.2998419. [Online]. Available: http://doi.acm.org/10.1145/2998415.2998419. 7

[25] K. Hazelwood and D. Grove, "Adaptive online context-sensitive inlining," in *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, Mar. 2003, pp. 253–264. DOI: 10.1109/CGO.2003.1191550. [Online]. Available: https://doi.org/10.1109/CGO.2003.1191550. 1, 2, 22, 24

[26] IBM and the Eclipse Foundation, *Eclipse openj9*, [Online; accessed January-21-2019], 2019. [Online]. Available: https://www.eclipse.org/openj9/. 7

[27] S. M. inc., *Java se hotspot vm*, [Online; accessed May-06-2019]. [Online]. Available: http://java.sun.com/javase/technologies/hotspot/. 4

[28] *Intel xeon platinum 8180 processor*, https://ark.intel.com/products/120496/Intel-Xeon-Platinum-8180-Processor-38-5M-Cache-2-50-GHz-, [Online; accessed 11-Februrary-2019]. 55

[29] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani, "A study of devirtualization techniques for a java just-in-time compiler," *SIGPLAN Not.*, vol. 35, no. 10, pp. 294–310, Oct. 2000, ISSN: 0362-1340. DOI: 10.1145/354222.353191. [Online]. Available: http://doi.acm.org/10.1145/354222.353191. 14, 15

[30] J. B. Kam and J. D. Ullman, "Monotone data flow analysis frameworks," *Acta Inf.*, vol. 7, no. 3, pp. 305–317, Sep. 1977, ISSN: 0001-5903. DOI: 10.1007/BF00290339. [Online]. Available: http://dx.doi.org/10.1007/BF00290339. 9, 10, 12

[31] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The java virtual machine specification-java se 8 edition, march 2014*. 4–6, 34, 39, 43, 44, 88, 8

[32] R. Mangal, M. Naik, and H. Yang, "A correspondence between two approaches to interprocedural analysis in the presence of join," in *Programming Languages and Systems*, Z. Shao, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 513–533, ISBN: 978-3-642-54833-8. 24

[33] M. Mohnen, "A graph—free approach to data—flow analysis," in *Compiler Construction*, R. N. Horspool, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 46–61, ISBN: 978-3-540-45937-8. 10

[34] K. Organization, *Kaffe*, https://github.com/kaffe/kaffe, 2011. 4

[35] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95, San Francisco, California, USA: ACM, 1995, pp. 49–61, ISBN: 0-89791-692-1. DOI: 10.1145/199448.199462. [Online]. Available: http://doi.acm.org/10.1145/199448.199462. 10, 25

[36] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek, "Appropriate statistics for ordinal level data: Should we really be using t-test and cohen'sd for evaluating group differences on the nsse and other surveys," in *annual meeting of the Florida Association of Institutional Research*, 2006, pp. 1–33. 64

[37] B. G. Ryder and M. C. Paull, "Elimination algorithms for data flow analysis," *ACM Comput. Surv.*, vol. 18, no. 3, pp. 277–316, Sep. 1986, ISSN: 0360-0300. DOI: 10.1145/27632.27649. [Online]. Available: http://doi.acm.org/10.1145/27632.27649. 10

[38] M. Sagiv, T. Reps, and S. Horwitz, "Precise interprocedural dataflow analysis with applications to constant propagation," in *TAPSOFT '95: Theory and Practice of Software Development*, P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 651–665, ISBN: 978-3-540-49233-7. 10, 25

[39] R. W. Scheifler, "An analysis of inline substitution for a structured programming language," *Commun. ACM*, vol. 20, no. 9, pp. 647–654, Sep. 1977, ISSN: 0001-0782. DOI: 10.1145/359810.359830. [Online]. Available: http://doi.acm.org/10.1145/359810.359830. 1, 21, 23

[40] A. Sewe, J. Jochem, and M. Mezini, "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*, ser. SPLASH '11 Workshops, Portland, Oregon, USA: ACM, 2011, pp. 317–328, ISBN: 978-1-4503-1183-0. DOI: 10.1145/2095050.2095102. [Online]. Available: http://doi.acm.org/10.1145/2095050.2095102. 23

[41] A. Shankar, M. Arnold, and R. Bodik, "Jolt: Lightweight dynamic analysis and removal of object churn," *SIGPLAN Not.*, vol. 43, no. 10, pp. 127–142, Oct. 2008, ISSN: 0362-1340. DOI: 10.1145/1449955.1449775. [Online]. Available: http://doi.acm.org/10.1145/1449955.1449775. 2, 21–23

[42] M. Sharir and A. Pnueli, *Two approaches to interprocedural data flow analysis*. New York, NY: New York Univ. Comput. Sci. Dept., 1978. [Online]. Available: https://cds.cern.ch/record/120118. 24

[43] O. G. Shivers, "Control-flow analysis of higher-order languages of taming lambda," UMI Order No. GAX91-26964, PhD thesis, Pittsburgh, PA, USA, 1991. 24

[44] D. Simon, J. Cavazos, C. Wimmer, and S. Kulkarni, "Automatic construction of inlining heuristics using machine learning," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, ser. CGO '13, Washington, DC, USA: IEEE Computer Society, 2013, pp. 1–12, ISBN: 978-1-4673-5524-7. DOI: 10.1109/CGO.2013.6495004. [Online]. Available: https://doi.org/10.1109/CGO.2013.6495004. 1, 2

[45] M. Torchiano, *Cliff.delta: Cliff's delta effect size for ordinal variables*, R Foundation for Statistical Computing, 2019. [Online]. Available: https://rdrr.io/cran/effsize/man/cliff.delta.html. 64

[46] Wikipedia contributors, *Java virtual machine — Wikipedia, the free encyclopedia*, [Online; accessed 4-June-2019], 2019. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Java_virtual_machine&oldid=899749630. 4

[47] X. Zhang, R. Mangal, M. Naik, and H. Yang, "Hybrid top-down and bottom-up interprocedural analysis," *SIGPLAN Not.*, vol. 49, no. 6, pp. 249–258, Jun. 2014, ISSN: 0362-1340. DOI: 10.1145/2666356.2594328. [Online]. Available: http://doi.acm.org/10.1145/2666356.2594328. 24

Appendix A

Java Bytecode Abstract Semantics

A.1 Transfer Functions

Category	CONTROLFLOW
Operation	Jump conditionally or unconditionally.
Description	The concrete instruction would normally jump to a different segment of code. However, because the flow of control is changed, when abstractly interpreting instructions that are supposed to change the flow of control, we only change elements on the stack and pass the current block's abstract state to the successor blocks. These instructions are also described more thoroughly in Section 5.2.

Category	CALLSTACK
Operation	Push onto call stack
Description	The concrete instruction would normally calls a method. These instructions are also described more thoroughly in Section 5.1. Dynamic methods are a not handled and are a source of imprecision. Future work includes handling correctly dynamic invocation instructions.

Category	ALOAD
Operation	Load from array.
Format	ALOAD
Operand Stack	*...,arrayref,index*→ *...,value*
Description	The *arrayref* must be a valid abstract value of the abstract array domain. The *index* must be a valid abstract value of the integer interval domain. The *value* placed in the operand stack is an abstract value of the type corresponding to the type of the concrete instruction. For example, if the concrete instruction is *iaload*, then *value* is of the integer interval domain type.
Note	If the *arrayref* is `null` in the concrete instruction semantics, then ALOAD throws a `NullPointerException`. In future work, method summaries can be extended with an entry to check for the `null`-ness of *arrayref*. The current implementation of creating method summary entries does not take *arrayref*'s `null`-ness into account. If *index* is not within the bounds imposed by the length of *arrayref*, in the concrete instruction semantics, then ALOAD throws a `ArrayIndexOutOfBoundsException`. In future work, method summaries can be extended with an entry to check the bounds of *index*. The current implementation of creating method summary entries does not take the bounds of *index* into account.

Category	ASTORE
Operation	Store into array.
Format	ASTORE
Operand Stack	*...,arrayref,index,value*→ *...*
Description	The *arrayref* must be a valid abstract value of the abstract array domain. The *index* must be a valid abstract value of the integer interval domain. The *value* must be a valid abstract value capable of being stored in *arrayref* For example if the concrete instruction is *iaload* then *value* is of the integer interval domain type.
Note	If the *arrayref* is `null` in the concrete instruction semantics, then ALOAD throws a `NullPointerException`. In future work, method summaries can be extended with an entry to check for the `null`-ness of *arrayref*. The current implementation of creating method summary entries does not take *arrayref*'s `null`-ness into account. If *index* is not within the bounds imposed by the length of *arrayref*, in the concrete instruction semantics, then ALOAD throws a `ArrayIndexOutOfBoundsException`. In future work, method summaries can be extended with an entry to check the bounds of *index*. The current implementation of creating method summary entries does not take the bounds of *index* into account. If the type of *value* is not assignment compatible with the concrete type component of *arrayref* in the concrete instruction semantics, then STORE throws an `ArrayStoreException`. In future work, method summaries can be extentended with an entry to check the class of *value*. The current implementation of creating method summary entries does not take the class of *value* into account.

Category LOAD

Operation Load from local variable array.

Format

LOAD
index

or

LOAD

Operand Stack ...→
...,*value*

Description The *index* is an unsigned byte that must be an index into the local abstract variable array of the current abstract frame. The local variable at *index* must contain an abstract value. The *value* in the local abstract variable at *index* is pushed onto the abstract operand stack.

Note This instruction may be affected if preceded by the *wide* instruction.

Category STORE

Operation Store element into local abstract variable

Format

STORE
index

or

STORE

Operand Stack ...,*value*→
...

Description The *index* is an unsigned byte that must be an index into the local abstract variable array of the current abstract frame. The *value* at the top of the abstract operand stack must be an abstract value. It is popped from the abstract operand stack, and the value of the local abstract variable at *index* is set to *value*.

Note This instruction may be affected if preceded by the *wide* instruction.

Category	RETURN
Operation	Return *value* from method
Format	RETURN
Operand Stack	*...,value*→ [empty]
Description	The *value* must refer to an abstract type that is assignment compatible with the return type of the current method.
Notes	Because upwards propagation in the IDT is not modelled, *areturn* is equivalent to *pop* whenever there is a return value. The values returned may be used in an entry in the method summary in a future implementation. This would allow our analysis to propagate information up the IDT. However, that is not the case in the current implementation of the abstract interpreter. The operand stack after the interpretation of these instructions is equivalent to being empty. In reality, the operand stack is not emptied, but it is no longer propagated.

Category	ABINOP
Operation	Perform arithmetic operations with abstract values.
Format	ABINOP
Operand Stack	*...,value1,value2*→ *...,result*
Description	Both *value1* and *value2* must be abstract values where the operation determined by the concrete instruction is compatible with the operands. The values are popped from the operand stack. The *result* is a safe approximation to the arithmetic operation of *value1* and *value2*. The *result* is pushed onto the operand stack.

Category	LBINOP
Operation	Perform logic operations with abstract values.
Format	LBINOP
Operand Stack	*...,value1,value2*→ *...,result*
Description	Both *value1* and *value2* must be abstract values where the operation determined by the concrete instruction is compatible with the operands. The values are popped from the operand stack. The *result* is a safe approximation to the logic operation of *value1* and *value2*. The *result* is pushed onto the operand stack.

Category	NEG
Operation	Negate abstract value.
Format	NEG
Operand Stack	*...,value*→ *...,result*
Description	The *value* must be an abstract value compatible with the negation operator. The *value* is popped from the operand stack. The *result* is a safe approximation to the negation of *value*. The *result* is pushed onto the operand stack.

Category	IINC
Operation	Increment local variable by constant
Format	*iinc* *index* *const*
Operand Stack	No change
Description	The *index* is an unsigned byte that must be an index into the local variable array of the current frame. The *const* is an immediate signed byte. The local abstract variable at *index* must contain an abstract value belonging to the abstract integer interval domain. The value *const* is first sign-extended to an `int` and then the local variable at *index* is incremented by that amount.
Notes	This instruction may be affected by the *wide* instruction.

ldc

Category	LDC
Operation	Push item from constant pool
Format	*ldc* *index*
Operand Stack	*...*→ *...,value*
Description	The *index* is an unsigned byte that must be an index into the runtime constant pool of the current class. The *value* in the constant pool at *index* is converted to an abstract value. The *value* is pushed onto the operand stack. If the value cannot be resolved statically, $\top$ is pushed onto the operand stack.

Category	CONST
Operation	Push immediate to stack
Format	CONST or CONST *value*
Operand Stack	*...,*→ *...,value*
Description	Cast immediate or value into an abstract value. Push value to abstract operand stack.

Category	CMP
Operation	Compare two abstract numeric types.
Format	CMP
Operand Stack	*...,value1,value2*→ *...,result*
Description	Both *value1* and *value2* must be of numeric type. The values are popped from the operand stack. An abstract value assignment compatible with the type of the concrete instruction is placed in the operand stack.

Category	STACK
Operation	Manipulate the stack
Description	Manipulates the stack according to the concrete semantics.
Note	Because of the wide diverse of formats available for this group of instructions, please consult the Java Virtual Machine Specification Manual [31, Chatper 6].

Category	MONITOR
Operation	Enter or exit monitor object
Format	MONITOR
Operand Stack	*..., objectref* → *...*
Description	Because our analysis does not model monitors, only the *objectref* is popped off the abstract stack.
Note	The concrete instruction semantics for *monitorenter* and *monitorexit* state that if *objectref* is `null`, the *athrow* instruction throws a `NullPointerException`. If the abstract interpreter is able to determine safely that *objectref* is either `null` or not `null` then a method entry can be added to the method summary. The current implementation does not consider this case.

Category	ATHROW
Operation	Throw exception of error
Format	*athrow*
Operand Stack	*..., objectref* → *objectref*
Description	Because our analysis does not model exceptional control flow, only the *objectref* is popped off the abstract stack.
Note	The concrete instruction semantics for *athrow* state that if *objectref* is `null`, the *athrow* instruction throws a `NullPointerException`. If the abstract interpreter is able to determine safely that *objectref* is either `null` or not `null` then a method entry can be added to the method summary. The current implementation does not consider this case.

Category GET

Operation Get from class or object

Format

GET
indexbyte1
indexbyte2

Operand Stack *...,[objectref]→*
...,value

Description Because this analysis is not field sensitive, the *value* pushed to the operand stack is always `null`.

Note The concrete instruction semantics for *getfield* mention that if *objectref* is `null`, the *getfield* instruction throws a `NullPointerException`. If the abstract interpreter is able to determine safely that *objectref* is either `null` or not `null` then a method entry can be added to the method summary. The current implementation does not consider this case.

Category PUT

Operation Set field in class or object

Format

PUT
indexbyte1
indexbyte2

Operand Stack *...,[objectref],value→*
...,value

Description Because this analysis is not field sensitive, this operation is equivalent to simply popping the values from the abstract operand stack.

Note The concrete instruction semantics for *putfield* mention that if *objectref* is `null`, the *putfield* instruction throws a `NullPointerException`. If the abstract interpreter is able to determine safely that *objectref* is either `null` or not `null` then a method entry can be added to the method summary. The current implementation does not consider this case.

Category WIDE

Operation Use a wide index to access the local variable array or the constant pool [31, Chapter 6].

Table A.1: Summary of JVM bytecodes

Category	Instructions
ALOAD	*aaload*, *baload*, *caload*, *daload*, *faload*, *iaload*, *laload*, *saload*
ASTORE	*aastore*, *bastore*, *castore*, *dastore*, *fastore*, *iastore*, *lastore*, *sastore*
LOAD	*aload*, *aload_<n>*, *dload*, *dload_<n>*, *fload*, *fload_<n>*, *iload*, *iload_<n>*, *lload*, *lload_<n>*
STORE	*astore*, *astore_<n>*, *dstore*, *dstore_<n>*, *fstore*, *fstore_<n>*, *istore*, *istore_<n>*, *lstore*, *lstore_<n>*
RETURN	*areturn*, *dreturn*, *freturn*, *ireturn*, *lreturn*, *ret*, *return*
ABINOP	*dadd*, *fadd*, *iadd*, *ladd*, *ddiv*, *fdiv*, *idiv*, *ldiv*, *dmul*, *fmul*, *imul*, *lmul*, *drem*, *frem*, *irem*, *lrem*, *dsub*, *fsub*, *isub*, *lsub*,
NEG	*dneg*, *fneg*, *ineg*, *lneg*
IINC	*iinc*
CAST	*checkcast*, *instanceof*, *d2f*, *d2i*, *d2l*, *f2d*, *f2i*, *f2l*, *i2b*, *i2d*, *i2f*, *i2l*
LDC	*ldc*, *ldc_w*, *ldc2_w*
CONST	*bipush*, *sipush*, *aconst_null*, *dconst_<d>*, *fconst_<f>*, *iconst_<i>*, *lconst_<l>*
CONTROLFLOW	*goto*, *goto_w*, *if_acmp<cond>*, *if_icmp<cond>*, *if<cond>*, *ifnonnull*, *ifnull*, *lookupswitch*, *tableswitch*
CMP	*dcmp<op>*, *fcmp<op>*, *lcmp*
LBINOP	*iand*, *land*, *ior*, *lor*, *ishl*, *lshl*, *ishr*, *lshr*, *iushr*, *ixor*, *lxor*
STACK	*dup*, *dup_x1*, *dup_x2 dup2*, *dup2_x1*, *dup2_x2*, *nop*, *pop*, *pop*2, *swap*
CALLSTACK	*jsr*, *jsr_w*, *invokedynamic*, *invokeinterface*, *invokespecial*, *invokestatic*, *invokevirtual*
MONITOR	*monitorenter*, *monitorexit*
ATHROW	*athrow*
GET	*getfield*, *getstatic*
PUT	*putfield*, *putstatic*
WIDE	*wide*