Efficiently Estimating and Exploiting the Indirect Benefits of Inlining

Master-Thesis von Jannik Jochem
Mai 2011

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Fachgebiet Softwaretechnik

Efficiently Estimating and Exploiting the Indirect Benefits of Inlining

Vorgelegte Master-Thesis von Jannik Jochem

1. Gutachten: Prof. Dr.-Ing. Mira Mezini
2. Gutachten: Dipl.-Math. Andreas Sewe

Tag der Einreichung:

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 30. Mai 2011

(Jannik Jochem)

Abstract

Inlining is a ubiquitous compiler optimization that significantly improves execution performance. When deciding whether to inline, gained runtime performance has to be weighed against increased compile time and program size. The performance benefit of inlining consists of a direct benefit from removing the call overhead, as well as an indirect benefit that may occur when inlining makes further optimizations possible. In state-of-the-art Java VMs, this indirect benefit is estimated only on the basis of static knowledge available at the call site. A particular kind of indirect benefit is the elimination of guard tests when further inlining is performed. Current inlining heuristics encourage inlining by assigning bonuses when static knowledge that may enable guardless inlining is present at the call site. In this thesis, it is shown that in the largest part of these cases, no further inlining actually occurs. When this is the case, assigning bonuses is unadvised and increases compile time. A heuristic is developed that uses readily-available call-graph profiles to predict when further inlining occurs. This heuristic is evaluated in terms of prediction quality and performance impact. The results show that the proposed heuristic decreases the risk of assigning unadvised bonuses. Measuring the performance effect of the proposed heuristic shows a reduction in overall execution time of 1 % to 5 % for some benchmarks, while execution time is increased by 2 % for a single benchmark.

Abstract (Deutsch)

Inline-Ersetzung ist ein in modernen Compilern häufig verwendetes Optimierungsverfahren, das die Performanz von Programmen stark verbessert. Um zu entscheiden, ob eine Inline-Ersetzung durchgeführt werden soll, wird der Nutzen dieser Operation gegen die Erhöhung von Kompilierzeit und Programmgröße abgewägt. Der Nutzen von Inline-Ersetzungen besteht aus einem direkten Nutzen durch das Wegfallen des Methodenaufrufs und des damit verbundenen Mehraufwands, sowie aus einem indirekten Nutzen durch die Ermöglichung weiterer Optimierungen. In modernen Java-Ausführungsumgebungen wird der indirekte Nutzen von Inlining anhand von statischen Informationen an der Aufrufstelle abgeschätzt. Eine besondere Art von indirektem Nutzen ist die mögliche Vermeidung von Guard-Tests bei weiteren Inline-Ersetzungen. Aktuelle Inline-Heuristiken vergeben Boni, wenn an der Aufrufstelle zusätzliche statische Informationen vorhanden sind, welche die Vermeidung von Guard-Tests ermöglichen könnten. Im Rahmen dieser Arbeit wird jedoch gezeigt, dass weitere Inline-Ersetzung in den meisten solcher Fälle nicht auftritt. Daher ist es selten sinnvoll, einen Bonus zu vergeben. Aus der ungerechtfertigten Vergabe von Boni folgt ein zu starker Einsatz von Inline-Ersetzung, der die Kompilierzeit des ausgeführten Programms erhöht. In dieser Arbeit wird eine Heuristik vorgestellt, welche den Aufrufgraphen, der ohnehin von der Laufzeitumgebung per Profilierung gewonnen wird verwendet, um das Auftreten von weiterer Inline-Ersetzung vorherzusagen. Diese Heuristik wird im Hinblick auf ihre Vorhersagequalität und ihre Auswirkungen auf die Performanz des ausgeführten Programms untersucht. Die Ergebnisse zeigen, dass die Heuristik das Risiko herabsetzt, ungerechtfertigt Boni zu vergeben. Außerdem erreicht die Heuristik eine Reduktion der Gesamtlaufzeit um 1 % bis 5 % für einige Benchmarks, während für einen einzelnen Benchmark eine Erhöhung der Gesamtlaufzeit um 2 % beobachtet wird.

Contents

1 Introduction

Object-oriented programming provides benefits in terms of understandability, reusability and maintainability of code. A well-designed object-oriented program typically consists of a large number of small methods that interact to provide the overall functionality of the program. This makes the individual methods easier to comprehend by the programmer. However, this benefit of object-oriented programming comes at a price. Invoking methods causes a significant runtime overhead that can severely impact execution performance. To remedy this situation, compilers of object-oriented languages use *method inlining* to remove the call overhead for some call sites.

Method inlining works by replacing a method *call site* inside a *caller method* with the body of the *callee method*. In this way, the call overhead for that particular call site is eliminated, improving the execution speed of the program.

The cost of inlining is two-fold. Firstly, replacing the call site with the body of the callee method means that a copy of the callee's code is created, which increases code size. Secondly, the additional code has to be processed by later compiler stages, which increases compile time. For statically-compiled programs, only the increase in code size is relevant for execution performance. In programs that are compiled just-in-time, increasing compile time also increases the total execution time of the program. This leads to a situation where the benefit of inlining has to be weighed against the additional cost incurred by inlining a particular call site. Compilers use an *inlining heuristic* to make a *cost-benefit tradeoff* for each call site that can potentially be inlined.

In addition to the problem of selecting the most profitable call sites to inline, the application of method inlining is complicated by the fact that many call sites in object-oriented programs are dispatched *virtually*. Virtual call sites dispatch to a *dynamic target method* according to the *dynamic type of the receiver*. This means that at the time when a call site is processed by the compiler, it is often impossible to determine the correct implementation of the callee method. If this is the case, the call site cannot be inlined directly [DA99].

Modern execution environments profile call sites to determine the *dynamic targets* and their distribution for the call site. For some call sites, these profiles show that a single dynamic target method is dispatched to most of the time. If this is the case, it is often profitable to inline the call site *speculatively* [Gro+95]. However, this does not guarantee that the call site always dispatches to that method. This means that steps have to be taken to ensure correct dispatch of the call site.

In other cases, analyzing the class hierarchy of the receiver may allow the compiler to prove that there is only one possible target for a call site [DGC95]. However, *dynamic class loading* may change the class hierarchy of the receiver after the call site has been inlined. In this case, the assumption of a single possible target no longer holds and the generated code becomes incorrect.

To ensure correct dispatch when virtual call sites are inlined speculatively, *guard tests* [DA99] are generated into the code of the inlined call site. These tests ensure that the inlined code is only executed if it is the correct implementation for the dynamic receiver type. If this is not the case, the call is dispatched virtually as a fallback. Guard tests make speculative inlining possible, but they carry a disadvantage. The test code has to be generated by the compiler, which leads to an increase in compile time and code size. Also, the guard test has to be performed every time the inlined code is executed, which increases execution time.

In some cases, inlining a call site causes additional static information about the arguments to be propagated into the callee code. For example, when a method is invoked with a constant argument, inlining that method causes the argument to be substituted with its constant value inside the callee method. If the callee performs computations on the argument, the constant argument value may allow additional constant folding to be performed inside the callee method. If this happens inside the condition of a control structure, this may cause additional dead code elimination. In this case, inlining *enables interprocedural optimizations*, which is an *indirect benefit* of inlining [DC94].

The propagation of additional static information into the callee is not restricted to primitive arguments. In addition to array types, static information about object-reference arguments may also be propagated into the callee context. When the inlined callee invokes a method on such an argument, the additional static information may *enable guardless inlining* of that call site. This is the case when the *precise type* of the argument is known. In some other cases, the additional static information allows safe guardless inlining of a target method in the presence of dynamic class loading.

The indirect benefits of inlining have to be taken into account by the inlining heuristic. When inlining enables additional interprocedural optimizations, these optimizations result in smaller code that executes more efficiently. The optimizations reduce the cost of inlining in terms of code size and compile time [DC94]. Also, since execution speed is improved, the benefit of inlining is increased. A good inlining heuristic should take both aspects into account when deciding whether to inline a call site.

One way to factor the indirect benefits of inlining into the cost-benefit tradeoff is to always encourage inlining of a call site that has additional static information available for propagation into the callee context. However, this is inaccurate because the indirect benefit does not always materialize. In the case of constant primitive arguments, there is no benefit if the inlined callee performs no computations on the arguments. An indirect benefit for an object-reference argument only occurs if the callee actually invokes a method on that argument which is subsequently inlined.

Previous approaches to solve this problem work by inlining the callee experimentally to determine if an indirect benefit occurs [DC94]. Since this causes additional compile time, this approach is unsuited when just-in-time compilation is used.

The availability of online *call-graph profiles* in modern virtual machines motivates a different approach for determining whether an indirect benefit due to further inlining is likely to occur. When a method invokes another method on one of its object-reference arguments, this method invocation is likely to result in a call-graph edge. When deciding whether to inline a call site that has additional static information available for its object-reference arguments, the call-graph information of the callee can be queried for outgoing edges. If this yields an edge that may be caused by the callee invoking a method on its argument, it is more likely that an indirect benefit will occur than if no edge was observed. In the progress of this thesis, a profile-based heuristic for predicting indirect benefits was designed, implemented for the virtual machine Jikes RVM, and evaluated in terms of prediction quality and execution performance.

The rest of this thesis is structured as follows. Chapter 2 describes the details of performing method inlining as an optimization. This includes considerations regarding the cost-benefit tradeoff, profile-driven inlining, and inlining of virtual call sites. Additionally, the inlining heuristic employed by the virtual machine used for the implementation of the proposed heuristic is examined. Chapter 3 introduces some notation that allows a concise description of when indirect benefits due to guardless further inlining occur. Based on this, a heuristic is developed that predicts such benefits on the basis of dynamic call-graph profiles. Chapter 4 presents the results of a quantitative evaluation of the described problem and the proposed heuristic. First, the relative prevalence of object-reference arguments with additional static information is determined. A method for objectively assessing the quality of heuristics that predict further inlining is developed and used to evaluate the proposed heuristic. Finally, the effect of the proposed heuristic on performance is evaluated. Chapter 5 gives a summary of other research on the topic of inlining heuristics. In Chapter 6, a conclusions of the results of this thesis is reached and opportunities for future work are discussed

2 The Inlining Optimization

Inlining is the process of replacing a method call site with the body of the called method. This chapter discusses the use of inlining as a compiler optimization. Section 2.1 explains how virtual call sites can be inlined using guard tests, and under which circumstances these guard tests can be omitted. Section 2.2 describes how profile data can be used to improve inlining decisions. In Section 2.3, the cost and benefit of inlining are examined. Section 2.4 describes how the virtual machine used for implementing the heuristics in this thesis performs inlining.

Listing 2.1 shows an example Java class RectangleContainer that manages a list of rectangles. The method printTotalArea() computes and prints the total area of all rectangles in the list.

```
1  class RectangleContainer {
2    List<Rectangle> rectangles = ...
3
4    public static int computeArea(
5      Rectangle r) {
6      return rectangle.width *
7        rectangle.height;
8    }
9
10   public void printTotalArea() {
11     int totalArea = 0;
12     for (Rectangle rectangle:
13       rectangles)
14       totalArea +=
15         computeArea(rectangle);
16     System.out.println(
17       "Total area of rectangles: "
18       + totalArea);
19   }
20 }
```

Listing 2.1: Computing the total area of a list of rectangles

Although the design of Rectangle and RectangleContainer is not very object-oriented, abstracting the area computation into a separate method makes sense. Encapsulating the functionality into a method improves the program from a readability and maintainability point of view. However, this approach also affects the execution of the program.

Whenever the highlighted call site is executed, a stack frame needs to be created for the method call, the actual computation is performed, the return value is pushed back on the stack, and control is returned to printTotalArea(). In the case of the fairly trivial method computeArea(Rectangle), the overhead that is required to handle the method invocation probably exceeds the cost of the actual computation.

Inlining the marked call site allows the compiler to remove the additional cost that was incurred by the procedural abstraction of computeArea(Rectangle).

```
1  class RectangleContainer {
2    ...
3    public void printTotalArea() {
4      int totalArea = 0;
5      for (Rectangle rectangle:
6        rectangles)
7        totalArea +=
8          rectangle.width * rectangle.height;
9      System.out.println(
10       "Total area of rectangles: "
11       + totalArea);
12   }
13 ...
```

Listing 2.2: Inlining computeArea()

Listing 2.2 shows printTotalArea() after computeArea(Rectangle) has been inlined. Since the invoked method is static, there is only one possible target method for the call site and inlining is trivial. Inlining has allowed the compiler to transform the program into a form that can be executed faster. In this case, the call site is in the body of a **for** loop. This means that the total execution time of printTotalArea() has been reduced by the saved call overhead multiplied by the size of the iterated list of rectangles. This is a significant gain, considering the fact that the optimization is easy to perform in this situation.

2.1 Inlining Virtual Call Sites

Virtual call sites in object-oriented languages may, in theory, have any number of targets. They are generally *polymorphic*. However, a large fraction of virtual call sites dispatches to a single target most of the time [Gro+95].

In some instances, the compiler can statically prove a call site to be *monomorphic* by analyzing its class hierarchy [DGC95]. In a language like C++, this is guaranteed to work. In Java, the possibil-

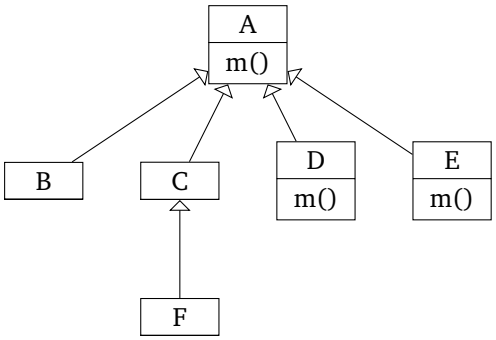

Figure 2.1: An example receiver class hierarchy [DA99]

ity of dynamic class loading complicates the situation. The compiler can prove monomorphism of a call site only for the current state of the receiver's class hierarchy. When a new class is loaded into the hierarchy, the call site may become polymorphic.

It is usually profitable to inline call sites that have a single dominant target. If the call site sometimes dispatches to another target method or if class loading can cause the call site to become polymorphic, it is necessary to *guard* the inlined call site against incorrect dispatch.

2.1.1 Guard Tests

The *class test* is one technique that ensures correct dispatch when inlining speculatively. Before the inlined callee is executed, the receiver's class is compared to the declaring class of the inlined method. If the classes match, the inlined code is executed. Otherwise, the call is dispatched virtually as a fallback. Listing 2.3 shows a Java implementation of the class test.

```
1  class Caller {
2    void method(A a) {
3      if (a.getClass() == A.class) {
4        // inlined body of A.m()
5      } else {
6        // off-branch fallback
7        a.m();
8      }
9    }
10 }
11
12 class A() {
13   void m() {
14     // body of A.m()
15   }
16 }
```

Listing 2.3: The class test

In Caller.method(A), a call site of the form A.m() was inlined using a class test. In line 3, the test is performed by determining whether the dynamic type of a is A. If this is the case, the inlined body of the method A.m() is executed. This branch of the test is called the *fast path*. If the class test fails, the *off-branch code* is executed to ensure correct dispatch.

In this example, a failed class test is handled by dispatching the call site virtually as shown in line 7. This is the normal way of handling failed guard tests. In Jikes RVM, the off-branch code may alternatively trigger recompilation of the root method (A.m(B) in this case), after which the executing method is replaced on-stack (cf. Section 2.1.2) with the recompiled version.

Detlefs and Agesen [DA99] argue that the class test is too restrictive in some situations. Figure 2.1 shows an example receiver class hierarchy. The method m() is defined in the super class A and is the correct implementation for the classes A, B, C and F. When inlining a call site that invokes m() on a receiver with the static type A, a class test has to be generated that covers all of the four classes for which A.m() is the correct method. This incurs a significant cost in code size and execution time on the fast path.

To remedy this situation, Detlefs and Agesen propose the *method test*. The method test requires each class to have a table of pointers to the method definitions that are valid for that class.

The method test works by first fetching the dynamic receiver's class object as in the class test. Then, the address of the correct implementation of the method to invoke is fetched from the class object's method table. If this is the same as the address of the inlined method, the inlined code is executed. Otherwise, the previously fetched method address is called. Listing 2.4 shows an example implementation of a method test using pseudocode. The pseudocode resembles Java and its reflection capabilities, except for the fact that checked exceptions are not caught for the sake of conciseness and a method literal is used in line 4.

```
1  class Caller() {
2    void method(A a) {
3      // compiler-inserted constant
4      Method inlinedTarget = A.m;
5
6      // fetch dynamic target
7      Method target =
8        a.getClass().getMethod("m()");
9
10     // perform the test
11     if (target == inlinedTarget) {
12       // inlined body of A.m()
13     } else {
14       // off-branch fallback
15       target.invoke(a);
16     }
17   }
18 }
19
20 class A() {
21   void m() {
22     // body of A.m()
23   }
24 }
```

Listing 2.4: The method test

The value of inlinedTarget identifies the method implementation that was inlined for the call site. When the method test is implemented in machine code, this is replaced by a constant that contains the method pointer of the inlined method. The value of target identifies the dynamic target of the method. Here, the reflective getMethod() operation is used to find the correct method reference. In a machine-code implementation, the method pointer is read from the dynamic receiver's method table at a constant offset that represents the virtual method to invoke. The actual test is performed by comparing the method reference of the inlined method to that of the dynamic target method. Note that the off-branch case invokes the dynamic target that was resolved as a part of the method test instead of performing a full virtual dispatch.

When the method test is performed, the class object of the dynamic receiver is fetched in the first step. In the second step, the method pointer for the correct implementation is fetched from the class object. Because of this, the method test requires two indirect read instruction as opposed to one indirect read instruction for the class test. However, when a number of classes in a hierarchy share the same implementation of a virtual method, a single method test is sufficient to cover all of those classes. In this case, the method test is more efficient than the class test.

The described guard tests have some shortcomings. A single class test cannot cover multiple dynamic receivers that share the same method definition. The code of a method test cannot be shared among multiple inlined methods that are called on the same receiver.

Arnold and Ryder [AR02] propose *thin guards*. A thin guard is implemented using a boolean flag that encodes assumptions that were made when the executing method was optimized. In the context of inlining, a guard flag is created that shows whether class loading has taken place since the method was optimized. When the method is executed, a single conditional operation can determine if the optimization assumptions are still valid. If this is the case, the inlined callee can be executed. Otherwise, the call has to be dispatched virtually.

Suppose that only the classes A, B and C from the class hierarchy in Figure 2.1 are loaded. Class hierarchy analysis proves that A.m() is the only valid target for a receiver of static type A. The compiler uses thin guards to generate the code in Listing 2.5.

```
1  class Caller() {
2    void method(A a) {
3      if (noClassLoadingHasOccured) {
4        // inlined body of A.m()
5      } else {
6        // off-branch fallback
7        a.m();
8      }
9    }
10 }
```

Listing 2.5: Using a thin guard

Here, noClassLoadingHasOccured is a thin guard flag that is initially set to **true**. As long as the value of the flag is **true**, it is safe to assume that the inlined implementation of m() is the only valid one. As soon as D or E are loaded, this assumption is no longer valid. When this happens, the boolean flag is set to **false** and the method reverts to using the slower off-branch code until it is recompiled.

Note that class loading does not generally lead to the call site becoming polymorphic. If the class F or any other class not part of the subtype hierarchy of A is loaded, the assumption that A.m() is the only valid implementation of m() is still true. In the presented example code, this will nevertheless cause the guard test to fail and revert to the slow path. However, it is possible to create thin guard flags for individual call sites or small sets of call sites that are only set to **false** when a *relevant* change to the class hierarchy occurs [AR02].

The thin guard test has the advantage that it can cover multiple call sites that dispatch on different receiver objects. However, thin guards can only be used to inline call sites that were proved *currently monomorphic*, meaning the call site was monomorphic when the optimization assumption was true.

This means that they cannot be used to inline polymorphic call sites at all.

2.1.2 Guardless Inlining

In some cases, it is possible to inline a virtual call site directly without having to use guards. In Java code, virtual call sites can be inlined safely without a guard if:

- the callee or its declaring class is **final**;
- the receiver's *precise type* is statically known; or
- the receiver *pre-exists* the invocation of the caller method.

A **final** declaration on the callee method or on the receiver class effectively renders the call site static, making guard tests unnecessary.

Most optimizing compilers perform some form of data-flow analysis on the methods they compile. This may allow the compiler to determine the *precise type* of a receiver in some cases. Listing 2.6 shows a method with a call site for which the precise type of the receiver is known.

```
1 class A {
2   void m() {
3     B b = new B();
4     b.n();
5   }
6 }
```

Listing 2.6: A call site with a precisely typed receiver

Here, it is obvious that b is always of type B. This property can be proved by a very simple data-flow analysis. Since b is known to always be of type B, the call site can be inlined without requiring a guard test.

In other cases where the receiver is not precise, inspecting the class hierarchy of the receiver may reveal that there is only a single possible target method for a call site [DGC95]. However, when a new class is loaded into the receiver's class hierarchy, the call site may become polymorphic. If this happens, the method that contained the original call site becomes invalid and needs to be recompiled.

This is problematic if the method in question is still executing. Since it is not guaranteed that the method will return in finite time, the guardlessly inlined call site may result in incorrect dispatch. If *currently monomorphic* call sites are inlined speculatively, this problem needs to be addressed.

One possible solution is to perform *on-stack replacement (OSR)* [HCU92] to transfer execution from the now-invalid method to a recompiled version. This technique requires the method's computational state to be encoded in such a way that the method can resume execution after the replacement has taken place. This introduces additional complexity to the generated code and restricts the applicability of optimizations [DA99].

Code patching [CLS00] solves the same problem in a different manner. This technique preemptively inserts initially disabled guard tests into the compiled method. When the optimization assumption is invalidated, a single instruction in the compiled method is overwritten to enable the guard, allowing safe execution of the method. This has the disadvantage that guard code is generated, which nullifies the compilation time and code-size benefits of direct inlining.

On-stack replacement and code patching are not required for all currently-monomorphic call sites. Detlefs and Agesen [DA99] introduce the concept of *pre-existence*. If the receiver of a call site is guaranteed to have been defined before the caller was invoked, it is said to *pre-exist the caller's invocation*. This is the case when the receiver was passed to the caller as an argument that is never overwritten during the execution of the method. If, on the other hand, the receiver is overwritten with an object, for example one returned from another method, this object may be of a dynamically-loaded class that did not exist in the system when the method started executing. In this case, the current execution of the method may be affected by class loading. If the receiver pre-exists the method invocation, this means that while the method is still executing, the receiver will never be an instance of a newly-loaded class. Therefore, all assumptions about the class hierarchy that were made when the caller was compiled are still valid while the caller finishes executing. When class loading occurs, correct dispatch is ensured by recompiling the method and using that recompiled version for all new invocations of the method.

2.2 Profile-Directed Inlining

Recall the example program in Listing 2.1 that computes the area of a number of rectangles. When the requirements for the program change in such a way that the total area has to be computed for a list of arbitrary shapes, the example may be refactored into a more object-oriented style as shown in Listing 2.7, which is based on an example by Grove et al. [Gro+95].

```
1  abstract class Shape {
2    public int area();
3  }
4
5  class Rectangle extends Shape {
6    int width, height;
7    public int area() {
8      return width*height;
9    }
10 }
11
12 class Circle extends Shape {...}
13
14 class ShapeContainer {
15   List<Shape> shapes = ...
16
17   public void printTotalArea() {
18     int totalArea = 0;
19     for (Shape shape: shapes)
20       totalArea += shape.area();
21     System.out.println(
22       "Total area of shapes: "
23       + totalArea);
24   }
25 }
```

Listing 2.7: Computing the total area of a list of shapes (refactored)

The abstract base class Shape declares the public method area() which computes the area of the shape. The Rectangle class extends Shape and implements the area() method with the correct area computation for rectangles.

The additional flexibility gained by employing object-oriented design here comes at a price. The formerly static call site in the loop that computes the total area has been replaced by a virtual call site. The static receiver of the call site is of type Shape. Since there are multiple subclasses of Shape that define area(), it is not clear from the static context of the call site which version of the method is best inlined, if any at all.

Grove et al. [Gro+95] have analyzed the receiver distribution for virtual call sites in a number of object-oriented programs written in C++ and Cecil. Their finding is that a large number of call sites (36 % for the C++ programs and 50 % for the Cecil programs) have a single dynamic target method. In addition to this, they find that for most polymorphic call sites, a single dynamic target method is clearly dominant.

In the example presented in Listing 2.7, it might be the case that most Shapes in the list of shapes for which the total area has to be computed are in fact Rectangles. If there is profile information available that supports this hypothesis, a compiler may decide to inline the dynamic target that is invoked most of the time. Listing 2.8 shows the resulting version of the method printTotalArea().

```
1  public void printTotalArea() {
2    int totalArea = 0;
3    for (Shape shape: shapes) {
4      if (shape.getClass()
5          == Rectangle.class) {
6          Rectangle r =
7            (Rectangle) shape;
8        totalArea +=
9          r.width * r.height;
10     } else {
11       totalArea += shape.area();
12     }
13   } System.out.println(
14     "Total area of shapes: "
15     + totalArea);
16 }
```

Listing 2.8: Inlining the dominant target method

In lines 4-5, the optimized version of the method determines if the dynamic type of shape is Rectangle by performing a class test. If this is the case, the inlined code for Rectangle.area() is executed. Otherwise, the call site is dispatched virtually. If a large percentage of the shapes processed are of type Rectangle, this kind of optimization is profitable.

While Grove et al. [Gro+95] performed their study of call site distributions using offline data, online profile-directed inlining is commonly used in virtual machines. In Jikes RVM, an adaptive optimization system [Arn+00b] uses live profile data to selectively optimize hot methods. In addition to this, the technique of feedback-directed inlining is employed. For this, a *call-graph profile* is captured by sampling caller-callee pairs in method prologues. The resulting call-graph profile contains call edges with attached execution frequencies. An *adaptive inlining organizer* periodically selects particularly hot call edges and marks them to be inlined once the caller is recompiled. If recompiling the caller would lead to the edge being inlined, the caller is queued for recompilation. This leads to often-executed call edges being progressively inlined once enough profile information is available.

2.3 The Cost and Benefit of Inlining

In the beginning of this chapter, Listing 2.1 was used to motivate the need for inlining. The subsequent sections discussed the peculiarities of inlining in an object-oriented execution environment with profile-directed optimizations. This section provides a breakdown of the cost that is incurred by inlining optimization as well as the benefit provided

by inlining. This serves as the basis for deciding whether inlining a particular call site is profitable or not.

2.3.1 The Cost of Inlining

The cost of inlining depends on whether it becomes necessary to compile additional code. When a particular callee method is called only from a single caller method in the program, inlining that method means that the callee does not have to be compiled as an independent, non-inlined method a second time. On the other hand, if there is a single additional caller of the program into which the callee is not inlined, a standalone version of the callee needs to be compiled. When the callee is inlined into multiple methods, the inlined callee code needs to be compiled once for each of those callers. This means that if there are at least two callers, of which at least one inlines the callee, inlining results in code duplication. Determining the number of potentially inlined call sites in a program requires statically analyzing the entire program beforehand. If this is not feasible, it is necessary to always assume that inlining a particular call site causes the callee code to be duplicated and therefore incurs a cost.

When static compilation is used, time spent compiling is rarely an issue. Because of this, the only relevant cost incurred by inlining is that of increased code size due to code duplication. Increasing the code size may have negative consequences for cache efficiency. Otherwise, code size is only an issue when available memory is constrained, for example on embedded systems.

When just-in-time compilation [Ayc03] is used, the time spent compiling is part of the overall execution time of the program. Code duplication caused by inlining increases compile time, which in turn increases execution time. Because of this, inlining many methods can have a strong negative effect on overall performance. For small methods, inlining saves compile time if the size of the inlined code is smaller than that of the call instruction that needs to be generated if the method is not inlined. If the size of the inlined code is greater than that of the call instruction, inlining causes additional compile time. Because of this, an accurate cost-benefit tradeoff has to be made when deciding whether to inline a particular call site.

2.3.2 The Benefit of Inlining

At the beginning of this chapter, the overhead of executing a call site was quoted as the prime benefit of inlining. The *call overhead* is composed of the following parts:

- creation of a stack frame for the callee method;
- the (virtual) dispatch of the target method; and
- returning control to the caller.

For small methods, this overhead is often greater than the cost of actually executing the callee method. This alone makes inlining a profitable optimization in those cases. For larger callee methods, eliminating the call overhead makes up a smaller part of the execution time for the callee method. In addition to removing the call overhead, inlining a method may also cause additional static information to be propagated from the caller method into the callee method. This additional static information may *enable additional optimizations*. An example of this is given in Listing 2.9.

```
1  static double root(double radicand,
2    int n) {
3    if (n == 2)
4      return Math.sqrt(radicand);
5
6    // compute root ...
7  }
8
9  static double length(int x, int y) {
10   return root(x*x + y*y, 2);
11 }
```

Listing 2.9: Calling a method with a constant argument

The method root() is a general utility method for computing the nth root of the double radicand. If n is 2, the default square root implementation is used. Otherwise, the root is computed using a numeric algorithm not detailed here. The method length() computes hypotenuse length using the pythagorean theorem. When the call of root() is inlined into length(), the constant argument is propagated into the context of the inlined callee as shown in Listing 2.10.

```
1  static double length(int x, int y) {
2    double radicand = x*x + y*y;
3
4    if (2 == 2)
5      return Math.sqrt(radicand);
6
7    // compute root
8    double result = ...
9
10   return result;
11 }
```

Listing 2.10: Propagating a constant into the callee context

After n has been substituted with the constant argument value, the compiler can apply constant fold-

ing [ASU07, p. 536]. Listing 2.11 shows the code after constant folding.

```
1  static double length(int x,
2    int y) {
3    double radicand = x*x + y*y;
4    double result;
5
6    if (true)
7      result = Math.sqrt(radicand);
8
9    // compute root
10   result = ...
11
12   return result;
13 }
```

Listing 2.11: The result of constant folding

By constant folding, the compiler determines that the expression inside the **if** statement is always true. In the next step, the unnecessary **if** statement is removed and dead-code elimination [ASU07, p. 553] is performed. Listing 2.12 shows the result of these optimizations.

```
1  static double length(int x,
2    int y) {
3    double radicand = x*x + y*y;
4
5    return Math.sqrt(radicand);
6  }
```

Listing 2.12: The inlined callee after optimization

In this example, the code of the inlined callee was subsequently reduced to the call to Math.sqrt(). The code for computing non-square roots was eliminated by the compiler. The result is a reduction in code size and compile time.

In this case, inlining root() enabled *interprocedural optimizations*. This kind of inlining benefit that is not related to the call overhead itself is called an *indirect benefit of inlining*. Common types of indirect benefit include:

- constant folding and consecutive dead-code elimination as shown in the example above;
- elimination of type checks;
- elimination of null checks;
- elimination of array bounds checks;
- optimizing array store operations by eliminating the need for type checks;
- guardless inlining due to precise types; and
- guardless inlining due to extant arguments.

This thesis is concerned with the last two types of benefit. When the precise type of an argument is known, methods invoked on that argument can be inlined without a guard because the target method is unambiguous. When an argument is extant, guardless inlining of a method invoked on that argument is possible if the call site is currently monomorphic.

2.4 Inlining in Jikes RVM

Jikes RVM is used for all experimentation in this thesis, including the example implementation of the heuristics proposed in Chapter 3. This section gives an overview of the aspects of Jikes RVM's implementation which are relevant to this work. This includes the adaptive optimization system and the just-in-time compilers as well as the inlining heuristic used by Jikes RVM.

2.4.1 Adaptive Just-In-Time Compilation

Jikes RVM compiles pure Java methods using two different compilers. Initially, methods are compiled using the *baseline compiler*, which has a high compilation rate, but produces code that is similar to bytecode interpretation in performance. An *adaptive optimization system* uses online profile data to select methods for recompilation using the *optimizing compiler* [Arn+00b]. Compiling methods with this compiler takes longer than with the baseline compiler but produces much faster code.

The optimizing compiler has three different *optimization levels* that employ progressively more complex optimizations.

Level 0 includes only optimizations that can be performed on-the-fly while transforming bytecode into Jikes RVM's high-level intermediate representation (HIR). Only small methods that do not require a guard are inlined.

Level 1 adds guarded inlining of call sites based on profile-directed inlining and a number of additional local and global optimizations.

Level 2 additionally uses static single assignment form (SSA) on scalar and array variables and some other optimizations.

Table 2.1 shows the performance metrics of the baseline compiler and the different optimization levels of the optimizing compiler. The data was measured with a Jikes RVM development version (SVN revision 16039) on the benchmark machine that was used for the performance measurements in Chapter 4. The data shown is the *Compiler DNA* [HGE10] of Jikes RVM, a set of performance metrics that is obtained using a specialized benchmark configuration of Jikes RVM. This data is used by the

Compiler	Compilation rate (bytecode bytes/ms)	Speedup factor
Baseline	875.98	1
Optimization level 0	24.37	5.81
Optimization level 1	14.08	7.02
Optimization level 2	13.73	7

Table 2.1: Performance metrics (Compiler DNA) of the compilers in Jikes RVM

adaptive optimization system to guide recompilation decisions.

The data shows that baseline compilation is about 35 to 70 times faster than optimized compilation, depending on the optimization level used. Recompiling at level 0 and level 1 yields large performance gains, while there is no additional benefit to be gained by recompiling at level 2.

2.4.2 The Inline Oracle

When a method is compiled by the optimizing compiler, Jikes RVM uses an *inline oracle* to decide whether a particular call site should be inlined. The oracle forms its decision based on the information available to the compiler when the call site is compiled. This information includes:

- the compiler settings to use;
- the static information from the caller's bytecode, like the static target of the call site and the static receiver type;
- dynamic profile data including the hotness of the call edge;
- additional information gained by performing data-flow analysis on the caller method, such as the precise type of the receiver; and
- a *method summary* of the callee including a size estimate of the method and other key characteristics, such as the presence of branch instructions or type checks in the method.

Using this information, the inline oracle decides which targets of the call site to inline (if any), which guard tests to use (if any) and whether to perform on-stack replacement instead of dispatching virtually when a guard test fails. The resulting decision is then implemented by the compiler.

The actual decision is reached in a five-step process.

1. Methods that should never be inlined are rejected.
2. If inlining the call site is trivial, a positive decision is reached and the oracle terminates.
3. If trivial inlining is not possible, the dynamic call graph is queried for the dynamic targets of the potentially polymorphic call site.
4. For each dynamic target, the oracle decides if inlining the target is desirable.
5. Guards are chosen for each desirable target.

In the first step, the oracle rejects inlining if the static callee is a native method, or the callee has a @NoInline annotation (used on some methods of the virtual machine). Inlining the constructor of java.lang.Throwable into any method that is not itself a constructor of Throwable is rejected as well.[1]

Then, trivial inlining is performed. An inlining operation is considered trivial precisely if the estimated size of the callee is below a predefined size limit and inlining it does not require a guard. This means that interface methods or methods declared abstract are never inlined in this step because they usually require a guard. Methods called by **invokevirtual** are inlined if they are declared final or defined by a final class. If the precise type of the call site's receiver has been determined by the compiler, this makes the call site provably monomorphic and it can be inlined trivially. If the call site is not provably monomorphic, a guard is required, which means that inlining the method is not a trivial operation. Additionally, callees annotated with an @Inline annotation are always inlined in this step.

When compiling at optimization level 0, only trivial inlining is performed. When a call site cannot be trivially inlined at optimization level 0, the oracle decides not to inline the call site at all. In all other optimization levels, the size of the root method that is being compiled is compared to a fixed threshold. If it is already larger than that threshold, inlining is also rejected.

If adaptive inlining is enabled and a dynamic call-graph profile is available, the oracle proceeds to query the dynamic call graph for the weight distribution of the sampled target methods at the call

[1] This is required because Jikes RVM captures a stack trace whenever it finds a Throwable constructor on the top of the call stack. Inlining this constructor would cause the constructor not to show up in the (unrestored) stack trace, in effect preventing the runtime from capturing the stack trace.

site in question. If the receiver type of the call site is precise, all targets but the one matching the precise type are discarded.

It is possible that there are no targets left after filtering. This can happen if the receiver type is precise, but none of the targets that were recorded in the call-graph profile match the precise type. In this case, the class hierarchy of the receiver is analyzed for possible targets. If the static target is an interface method or an abstract method, and has only a single implementation, that target is used as the basis for the inlining decision.

If there are no possible targets at that point, inlining the call site is rejected. Any targets that are already present in the inline sequence because of recursion or that are marked as non-inlineable by an annotation are rejected as well.

The possible targets are then analyzed separately to reach individual decisions. A target is inlined if it has an @Inline annotation or if it was inlined into the same method in previous runs of the optimizing compiler.

Otherwise, the oracle tries to estimate the cost of inlining by estimating the size of the inlined callee as well as that of any guards required for inlining it safely. Any target that does not account for at least 40% of the target weight distribution of the call site is rejected. The maximum allowable cost is adjusted if the profile data shows that the call edge is hot. Additionally, a penalty is attributed based on the current inlining depth to prevent too many levels of the call hierarchy from being inlined into the same root method.

Finally, the estimated cost is compared to the maximum allowable cost that was previously adjusted for edge temperature. If the estimated cost is less than or equal to the maximum cost, the target is inlined.

The inlining oracle uses an estimate of the inlined size of each target as the cost part of the cost-benefit tradeoff for all inlining decisions that are not preempted by annotations or other conditions that preclude inlining. For all methods, a size estimate is computed when its class is first loaded. This estimate is obtained by iterating the method bytecode, and is computed as the sum of the estimated machine code sizes of the individual instructions.

As shown in Section 2.3, inlining may cause additional static information about the arguments of the call site to be propagated into the callee context. This includes constant arguments of primitive and object type, precise type information for object arguments or array object arguments and pre-existence of object arguments. This information improves the opportunities for interprocedural optimization when it is propagated into the callee context. These optimizations result in a reduced code size of the inlined callee. Because of this, the presence of additional static information at the call site is factored into the cost-benefit tradeoff by modifying the inlined-size estimate of the callee.

The effect of interprocedural optimizations enabled by inlining on the cost-benefit tradeoff is not restricted to a size reduction on the cost side of the inlining operation. Reducing the amount of instructions that have to be executed in the inlined callee also causes a benefit in terms of execution speed. In Jikes RVM, these benefits are factored into the cost-benefit tradeoff only by reducing the cost. Since the cost-benefit tradeoff is represented as an inequation, reducing cost has the same effect on the final inlining decision as increasing the benefit. This means that the granted size bonus includes not only the actual size reduction in the callee, but also the benefit gained by improving execution efficiency. Therefore, the granted bonuses do not accurately reflect the actual size reduction of the callee but represent an abstracted overall benefit. The bonuses are arrived at by tuning, which means that they represent a notional size reduction that produces the best cost-benefit ratio for the set of benchmarks that was used for tuning.

The inline oracle analyzes each argument separately. Depending on the type of the argument and the additional static information provided by the caller, a constant size reduction is attributed. These bonuses are:

- 15% if the argument is a reference to an object of precise type;
- 5% if the argument pre-exists the invocation of the root method;
- 10% if the argument is **null**;
- 10% the argument is a string literal or another object constant;
- 5% if the argument is an integer constant;
- 5% if the argument is an object array of precise type; and
- 2% if the argument is an object array for which a type check is not required when performing a store operation and the method summary of the callee shows that the callee contains an array store operation.

These reductions are added and applied to the size estimate. The maximum reduction is limited to 40%. This means that the final size estimate is at least 60% of the estimate computed from the method bytecode.

Finally, the oracle adds the size estimate for any guards and off-branch code that is required for inlining the callee.

3 Proposed Solution

Inlining may propagate precise and extant arguments into the callee context, which can enable guardless further inlining of methods called on those arguments as an indirect benefit. This chapter introduces a notation for inlining decisions which can be used to concisely describe when such indirect benefits occur. A heuristic based on dynamic call-graph profiles is proposed that predicts when further inlining of methods called on precise or extant arguments is likely. This heuristic is used to modify the inlined-size estimate of inlining candidates. Additionally, two variants of the heuristic are proposed that may improve inlining decisions with regard to specific situations.

3.1 Problem Definition

When inlining a callee method with one or more precise or extant arguments, this additional static information is propagated into the callee context. When the callee invokes a method on one of these arguments, the corresponding call site can usually be inlined without requiring a guard test. These *indirect benefits* of inlining affect the cost-benefit tradeoff made when deciding whether to inline a call site. The objective of the work presented here is to efficiently *predict these benefits* in order to modify the cost-benefit equation adequately.

To allow concise reasoning about consecutive inlining of multiple methods into a single *root method*, some notation is introduced. Based on that notation, the problem of predicting indirect benefits caused by guardless inlining of methods invoked on precise or extant arguments is defined.

3.1.1 Inlining Decisions

Listing 3.1 shows a trivial example of an inlining decision. When the compiler processes the body of the method A.m(), it encounters the call site b.n().

```
1 class A {
2   void m() {
3     B b = ...
4     b.n();
5   }
6 }
```

Listing 3.1: A call site considered for inlining

At this point, the compiler needs to decide whether this call site should be inlined. This situation is called an *inlining problem*. If A.m() is not being inlined into another method itself, it is called a *root method*. Notation 3.1 represents the problem of inlining B.n() into A.m() by an arrow decorated with a question mark and should be read as "B.n() is being considered for inlining into the root method A.m()". In this notation, the callee that is shown is the static target method encoded in the call site.

Notation 3.1. A.m() $\overset{?}{\Leftarrow}$ B.n()

Should the compiler decide to inline the call site in question, an *inlining decision* is reached. If the compiler makes a *positive inlining decision*, the call site is inlined. The result is Notation 3.2, which can be read as "B.n() is inlined into A.m()". The shown callee is the actual target method inlined by the compiler, not the static target of the call site. In the example presented in Listing 3.1, the static target of the call site and the inlined target are the same method B.n().

Notation 3.2. A.m() $\Leftarrow$ B.n()

A *negative inlining decision* means that the target is not inlined. Notation 3.3 represents this case by diagonally striking out the box arrow and should be read as "B.n() is not inlined into A.m()".

Notation 3.3. A.m() $\not\Leftarrow$ B.n()

An inlined target method may contain a call to another method that is then also inlined. This is called *further inlining*. Notation 3.4 shows an abstract representation of such a case. A.m() is still the root method here, with B.n() inlined into A.m() and C.o() inlined into the inlined body of B.n().

Notation 3.4. A.m() $\Leftarrow$ B.n() $\Leftarrow$ C.o()

3.1.2 Class Hierarchies

Reasoning about virtual method calls in object-oriented programs makes a concise subtype notation necessary. Figure 3.1 shows a simple class hierarchy.

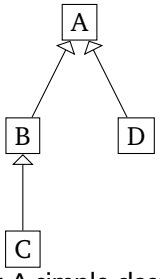

Figure 3.1: A simple class hierarchy

A subtype relation $a <: b$ can be defined that has the meaning "a is a subtype of b". It is useful to

define a strict version $a \ll: b \Leftrightarrow (a <: b) \wedge (a \neq b)$. Analogously, $a :> b$ is defined to mean "a is a supertype of b", along with a strict version $a :\gg b \Leftrightarrow (a :> b) \wedge (a \neq b)$. Some example subtype relations for the class hierarchy shown in Figure 3.1 are:

$$\begin{aligned} A <: A \\ \neg(A \ll: A) \\ B \ll: A \\ A :> B \\ B :\gg C \\ C <: A \\ \neg(D <: B) \end{aligned}$$

3.1.3 Callee Arguments

The target method of the call site to inline may be passed one or more arguments. In the context of this work, only scalar object-reference arguments are of interest. Therefore, primitive arguments and array arguments are excluded from the notation.

```
1  class A {
2    void m() {
3      B b = ...
4      C c = ...
5      b.n(c);
6    }
7  }
8  class B {
9    void n(C c) {...}
10 }
```

Listing 3.2: A call site with an object-reference argument

Listing 3.2 shows a callee with an object-reference argument. Notation 3.5 presents the inlining problem with the static type of the formal argument added.

Notation 3.5. A.m() $\overset{?}{\Leftarrow}$ B.n(C)

Here, only a single argument is shown. This is inaccurate because in the example shown in Listing 3.2, there is an implicit **this** argument that contains the dynamic receiver of the call. When further inlining of a method called inside B.n(C) takes place, there is no difference if the method is invoked on **this** or on any other argument. Therefore, no distinction is made in the notation presented here. In addition to implicit **this** arguments, methods may also have more than a single object-reference argument.

The notation that is presented here is used to reason about further inlining of methods that are invoked on an argument. In Java, methods have a single receiver and are dispatched only by that receiver. Because of this, there is no relevant interaction between multiple object-reference arguments of a method with regard to the problem at hand. In order to make predictions about further inlining of methods invoked on the arguments of a method, it is sufficient to inspect each argument in isolation. Therefore, the notation that is introduced here shows only a single argument. The observations and methods described here can be transferred to methods with multiple arguments by examining each argument in turn.

To illustrate the fact that some arguments are not shown, Notation 3.6 uses an ellipsis (...) to show where additional arguments would have been.

Notation 3.6. A.m(...) $\overset{?}{\Leftarrow}$ B.n(..., C, ...)

As described in Section 2.1.2, sometimes the *precise type* of an argument is known. Listing 3.3 shows such a case.

```
1  class A {
2    void m() {
3      B b = ...
4      C c = new D();
5      b.n(c);
6    }
7  }
8  class D extends C {...}
```

Listing 3.3: Inlining a callee with a precise argument

When B.n(C) is inlined into A.m(), the information that d is of the precise type D is propagated into the body of B.n(C). Notation 3.7 shows the precise type information along with the statically declared type. The meaning of this notation is that B.n(C) is inlined into A.m(), and by doing so the information that an argument of static type C has the precise type D is propagated into the callee.

Notation 3.7. A.m(...) $\Leftarrow$ B.n(..., D $\prec$ C, ...)

Note that the information is only propagated if inlining is actually performed. Therefore, this notation only makes sense with a positive inlining decision shown by an undecorated arrow.

If an argument's type is not precisely known, it is still possible that it *pre-exists* [DA99] the invocation of the root method. This means that its class is guaranteed to have been fully loaded before the root method was invoked. In Jikes RVM, arguments that have this property are called *extant*. In Listing 3.4, the argument of the call site in line 4 is extant because it is passed to the root method as an argument and is never assigned to.

```
1  class A {
2    void m(C c) {
3      B b = ...
4      b.n(c);
5    }
6  }
```

Listing 3.4: A call site with an extant argument

Notation 3.8 represents the case that B.n(C) was inlined into A.m(C), with the argument of type C extant in the caller A.m(C).

Notation 3.8. $\text{A.m}(\ldots) \Leftarrow \text{B.n}(\ldots, \tilde{\text{C}}, \ldots)$

Depending on the data flow in B.n(C), the argument may or may not have the extant property when a method is invoked on the argument. In general, once the argument or a copy of the argument is assigned to, the variable is very likely no longer extant. Otherwise, the argument is extant in the inlined body of B.n(C).

3.1.4 Precise and Extant Receivers

The dynamic receiver of a virtual call site is not generally known when compiling the caller. When such a call site is inlined, a guard test is often necessary to ensure correct dispatch. Inserting a guard test into the caller method incurs an overhead in code size and execution time.

```
1  class A {
2    void n() {
3      B b = new B();
4      b.o();
5    }
6  }
```

Listing 3.5: Direct inlining with a precise receiver

Listing 3.5 shows a call site where the receiver type is precisely known. Because of the precise type, the target of the call site is unambiguous and can be inlined without a guard. By omitting the guard, code size is reduced and less instructions need to be executed when the inlined call site is reached.

Direct inlining is also possible when there is only a single valid target method for the call site.

```
1  class B {
2    void n(C c) {
3      c.o();
4    }
5  }
6
7  abstract class C {
8    abstract void o();
9  }
10
11 class D extends C {
12   void o() {...}
13 }
```

Listing 3.6: Direct inlining with an extant receiver

Listing 3.6 shows an abstract class C with its single subtype D. D.o() is the only definition of that method across the entire class hierarchy. The compiler can prove that D.o() is the only valid target for the call site [DGC95].

The single target method can be inlined speculatively. However, dynamic class loading can invalidate the assumption that D.o() is the only target of the call site. Because the caller method may be executing at the time when dynamic class loading occurs and continue to do so indefinitely, on-stack replacement [HCU92] is usually required in these situations to ensure correct dispatch.

The argument c is extant in B.n(C). This means that if class loading takes place while B.n(C) is executing, it is guaranteed that the type of c does not become one of the newly-loaded types for the duration of the invocation. Therefore, it is sufficient to recompile B.n(C) and replace it for all new invocations of the method when classloading occurs.

On-stack replacement restricts the applicability of optimizations in the compiled method and incurs a cost when transferring the state of a currently executing method to the version it is being replaced by [DA99]. The fact that c is extant here saves this overhead while still allowing the call site to be inlined without a guard.

3.1.5 Inlining as an Enabling Transformation

The previous section has shown how precise and extant arguments can reduce the cost of inlining in terms of code size and execution speed. In some cases, the precise or extant arguments originate in the context of another method.

```
1  class A {
2    void m() {
3      B b = ...
4      C c = new D();
5      b.n(c);
6    }
7  }
8
9  class B {
10   void n(C c) {
11     c.o();
12   }
13 }
```

Listing 3.7: Precise type information in the caller context

Listing 3.7 shows a method A.m() with a precisely typed local variable c that is passed to a second method B.n(C) as an argument. When B.n(C) is compiled as a root method, the receiver of the call site c.o() is not precise.

However, if the call site in A.m() is inlined, the precise type information for c is propagated into the inlined body of B.n(C). This allows the call site c.o() to be inlined without a guard. Equation 3.1 shows how this can be expressed using the presented notation. Note that the notation shows the general form for methods with an arbitrary number of arguments.

$$\begin{aligned}&(\mathrm{A.m}(\ldots) \Leftarrow\!\!\!\!\exists\ \mathrm{B.n}(\ldots, \mathrm{D} \prec \mathrm{C}, \ldots))\\ \Rightarrow&(\mathrm{A.m}(\ldots) \Leftarrow\!\!\!\!\exists\ \mathrm{B.n}(\ldots, \mathrm{D} \prec \mathrm{C}, \ldots) \Leftarrow\!\!\!\!\exists\ \mathrm{D.o}(\ldots))\end{aligned} \tag{3.1}$$

For extant arguments, it is not immediately obvious how inlining can be interpreted as an enabling transformation. In Listing 3.8, c is extant in B.n(). This means that if B.n(C) is not inlined but is a root method itself, the call to c.o() can be inlined directly without the need for guards or later on-stack replacement. If B.n(C) is inlined, this is only possible if the argument is also extant in the root method.

```
1  class A {
2    void l() {
3      C c = ...
4      b.n(c);
5    }
6    void m( C c ) {
7      b.n(c);
8    }
9  }
10
11 class B {
12   void n(C c) {
13     c.o();
14   }
15 }
```

Listing 3.8: Extant type information in the caller context

To illustrate the more complex situation, Listing 3.8 shows two methods that both call and possibly inline B.n(C). In A.l(), the argument passed to B.n(C) is not extant, while in A.m(C) it is. When the call site in A.l() is inlined, the argument c is no longer extant in the entire body of the compiled method that results from inlining the call site. Because of this, inlining the call site in A.l() removes the extant benefit. Equation 3.2 shows the result of inlining in this situation.

$$\mathrm{A.l}() \Leftarrow\!\!\!\!\exists\ \mathrm{B.n}(\mathrm{C}) \Leftarrow\!\!\!\!\exists\ \mathrm{D.o}() \tag{3.2}$$

When A.m(C) is the root method, the extant benefit materializes and the last call site in the inlining sequence can be inlined directly as shown in Equation 3.3.

$$\mathrm{A.m}() \Leftarrow\!\!\!\!\exists\ \mathrm{B.n}(\tilde{\mathrm{C}}) \Leftarrow\!\!\!\!\exists\ \mathrm{D.o}() \tag{3.3}$$

In the first case, the benefit of the extant argument c was actually removed as a consequence of inlining. This raises the question whether in the second case, it is correct to say that the first step of the inlining sequence *enabled* the indirect benefit in the second step. It could be argued that the first inlining step merely failed to prevent the extant benefit.

Whether there is an indirect benefit to inlining B.n(C) into A.m() with an extant argument *depends* on the assumption that the first step of the inline sequence is inlined. It can be formulated as "if B.n(C) is inlined into A.m() with an extant argument, is it likely that an indirect benefit will occur when inlining a call on the argument?"

This is the information that the compiler needs when deciding whether to inline in the first place. In the first case where the indirect benefit does not materialize, the question is irrelevant because there is no extant argument to begin with. This is reflected by making the assumption of an extant argument explicit in Equation 3.4.

$$\begin{aligned}&(\mathrm{A.m}(\ldots) \Leftarrow\!\!\!\!\exists\ \mathrm{B.n}(\ldots, \tilde{\mathrm{C}}, \ldots))\\ \Rightarrow&(\mathrm{A.m}(\ldots) \Leftarrow\!\!\!\!\exists\ \mathrm{B.n}(\ldots, \tilde{\mathrm{C}}, \ldots) \Leftarrow\!\!\!\!\exists\ \mathrm{D.o}(\ldots))\end{aligned} \tag{3.4}$$

3.1.6 Further Inlining and its Benefits

When a compiler decides an inlining problem, it determines if the cost of inlining justifies the benefit of inlining. In a just-in-time compiler, the cost of inlining consists of both the additional machine code to be generated as well as the time it takes to compile the inlined method.

In Jikes RVM, the benefit of inlining is not directly estimated. Instead, the cost of inlining a particular call site is compared to a maximum allowable cost for the operation. This cost is fixed for trivial methods. For non-trivial methods, it is adapted according to the temperature of the call edge to inline as well as additional factors such as the inlining depth.

When inlining methods with precise or extant arguments, it makes sense to take the gained benefits for further inlining into account when making a cost-benefit tradeoff regarding the method to inline. The current technique used for this purpose in Jikes RVM is to grant a bonus on the size estimate for each precise or extant argument. This assumes that a method invokes at least one method on each of its object arguments and that those methods are subsequently inlined. This heuristic is crude at best.

This motivates the need for a formal definition as to when an indirect benefit caused by a precise or extant argument actually materializes. Whether a benefit is gained depends on whether a call to a precise or extant argument is present and is inlined. Definition 3.1 describes a predicate that determines if further inlining occurs for a particular inlining problem.

Definition 3.1. *A* further inlining predicate $\mathscr{P}$ *is a function that maps from an inlining problem* $\mathrm{A.m}(\ldots) \overset{?}{\Leftarrow} \mathrm{B.n}(\ldots)$ *to a boolean value. The predicate* $\mathscr{P}$ *determines if inlining the given problem causes further inlining. Equation 3.5 defines the value of* $\mathscr{P}$.

$$\begin{aligned} &\mathscr{P}(\mathrm{A.m}(\ldots) \overset{?}{\Leftarrow} \mathrm{B.n}(\ldots)) \\ &= (\exists \mathrm{Z})\big((\mathrm{A.m}(\ldots) \Leftarrow \mathrm{B.n}(\ldots)) \\ &\qquad \Rightarrow (\mathrm{A.m}(\ldots) \Leftarrow \mathrm{B.n}(\ldots) \Leftarrow \mathrm{Z.o}(\ldots))\big) \end{aligned} \tag{3.5}$$

Definition 3.1 covers all situations in which further inlining occurs. Given a general inlining problem, it determines if any further inlining takes place as a result of a positive inlining decision. To determine if further inlining of a method invoked on an extant argument occurs, Definition 3.1 is too coarse. Definition 3.2 provides a refined predicate for the case of further inlining on extant arguments.

Definition 3.2. *An* extant benefit predicate $\mathscr{P}_{extant} \subset \mathscr{P}$ *is a function that maps from an inlining problem of the form* $\mathrm{A.m}(\ldots) \overset{?}{\Leftarrow} \mathrm{B.n}(\ldots, \tilde{\mathrm{X}}, \ldots)$ *to a boolean value.* $\mathscr{P}_{extant}$ *determines if inlining the given problem causes a benefit through further inlining of a method with an extant receiver. Equation 3.6 defines the value of* $\mathscr{P}_{extant}$.

$$\begin{aligned} &\mathscr{P}_{extant}(\mathrm{A.m}(\ldots) \overset{?}{\Leftarrow} \mathrm{B.n}(\ldots, \tilde{\mathrm{X}}, \ldots)) \\ &= (\exists \mathrm{Y})\big(((\mathrm{X} <: \mathrm{Y}) \vee (\mathrm{Y} <: \mathrm{X})) \\ &\qquad \Rightarrow (\mathrm{A.m}(\ldots) \Leftarrow \mathrm{B.n}(\ldots, \tilde{\mathrm{X}}, \ldots)) \\ &\qquad \Rightarrow (\mathrm{A.m}(\ldots) \Leftarrow \mathrm{B.n}(\ldots, \tilde{\mathrm{X}}, \ldots) \Leftarrow \mathrm{Y.o}(\ldots))\big) \end{aligned} \tag{3.6}$$

Definition 3.2 describes a situation in which a method B.n(X) is inlined into a method A.m() with an extant argument of static type X. This causes an extant benefit when the method called on the extant argument inside B.n(X) is inlined.

Because the precise type of the argument is not known, the dynamic type of the argument may be X or any of its subtypes. The method that is actually inlined can be defined in any class Y that is on an inheritance path from the dynamic type of the argument to the root of the class hierarchy. This requirement is formalized as $(\mathrm{X} <: \mathrm{Y}) \vee (\mathrm{Y} <: \mathrm{X})$.

Definition 3.3 refines the further inlining predicate to determine when further inlining occurs on a precise argument.

Definition 3.3. *A* precise benefit predicate $\mathscr{P}_{precise} \subset \mathscr{P}$ *is a function that maps from an inlining problem of the form* $\mathrm{A.m}(\ldots) \overset{?}{\Leftarrow} \mathrm{B.n}(\ldots, \mathrm{Y} \prec \mathrm{X}, \ldots)$ *to a boolean value.* $\mathscr{P}_{precise}$ *determines if inlining the given problem causes a benefit through further inlining of a method with a precise receiver. Equation 3.7 defines the value of* $\mathscr{P}_{precise}$.

$$\begin{aligned} &\mathscr{P}_{precise}(\mathrm{A.m}(\ldots) \overset{?}{\Leftarrow} \mathrm{B.n}(\ldots, \mathrm{Y} \prec \mathrm{X}, \ldots)) \\ &= (\exists \mathrm{Z} :> \mathrm{Y}) \\ &\qquad \big((\mathrm{A.m}(\ldots) \Leftarrow \mathrm{B.n}(\ldots, \mathrm{Y} \prec \mathrm{X}, \ldots)) \\ &\qquad \Rightarrow (\mathrm{A.m}(\ldots) \Leftarrow \mathrm{B.n}(\ldots, \mathrm{Y} \prec \mathrm{X}, \ldots) \Leftarrow \mathrm{Z.o}(\ldots))\big) \end{aligned} \tag{3.7}$$

Definition 3.3 describes the situation where a precisely typed argument is propagated by inlining B.n(X) into A.m(). The precise type of the argument is Y. An indirect benefit occurs if a method that is called on the precise argument is inlined. This method is either defined or overridden in class Y or in any super type of Y. Therefore, the declaring class of the inlined method Z.o() is specified as $\mathrm{Y} <: \mathrm{Z}$ in the equation.

The further inlining predicate $\mathscr{P}$ and the refined predicates $\mathscr{P}_{\text{extant}}$ and $\mathscr{P}_{\text{precise}}$ are helpful for reasoning about the situations in which further inlining on extant or precise arguments occurs. However, computing them requires knowledge about the actual outcome of inlining decisions that are made after the initial inlining decision that is their input. This means that they can only be computed for a particular inlining problem after the root method of the inlining problem has been compiled. Therefore, it is not possible to use the outcome of the described predicates to guide inlining decisions based on indirect benefits. This motivates the need for a *further inlining prediction heuristic* that approximates the results of the predicate functions $\mathscr{P}_{\text{extant}}$ and $\mathscr{P}_{\text{precise}}$ as accurately as possible at the point in time when the initial inlining problem is decided by the compiler.

3.2 Solution Approach

Based on Definition 3.2 and Definition 3.3, a heuristic that predicts further inlining on precise and extant arguments can be defined. The heuristic predicts if, in the presence of a precise or extant argument, a call in which that argument is the receiver is likely to be inlined. If such a call is inlined depends on two conditions:

1. The method that is passed the argument calls a virtual method on that argument.
2. The inlining heuristic must actually decide to inline this call site.

If the first condition applies can be approximated using a *dynamic call graph profile*. Some virtual machines collect such a profile. In Jikes RVM, the inline oracle uses the call-graph profile to determine the weight distribution of targets when inlining call sites that are possibly polymorphic. Definition 3.4 gives a formal definition of a call-graph profile.

Definition 3.4. *A* call graph $CG = (N, E \subseteq N \times N)$ *is a multigraph with nodes N and edges E. A set of nodes N consists of individual method nodes* $N_i = T_i.m_i$*, where* m_i *is the signature of a method and* T_i *is the class that defines the method. A set of edges E consists of individual edges* $E_i = N_i \rightarrow N_j$ *that represent method* N_i *calling method* N_j*.*

A dynamic call graph $DCG \subset CG$ *is a subgraph of a program's call graph that contains only those call edges that are sampled at runtime.*

This is the definition given by Arnold and Grove [AG05], with type information for method receivers added and edge frequencies omitted.

The adaptive optimization system of Jikes RVM only selects frequently executed methods for recompilation with the optimizing compiler. This means that when the optimizing compiler decides whether to inline a particular method, it is very likely that the inlining candidate has already been executed a number of times. Therefore, call-graph information for the inlining candidate is likely to be available and reasonably accurate at this point in time.

When deciding whether to inline a candidate method, the dynamic call graph can be queried for outgoing call edges from that method. Consider the method B.l(C) shown in Listing 3.9.

```
1 class B {
2   void l(C c) {
3     c.o();
4   }
5 }
```

Listing 3.9: Invoking a method on the argument

When the compiler processes the inlining problem A.k() $\overset{?}{\Leftarrow}$ B.l(D $\prec$ C), querying the dynamic call graph for B.l(C) is likely to yield a call edge B.l(C) $\rightarrow$ D.o(). Because the argument of the method B.l(C) is known to be of precise type D, this edge strongly suggests that there is an invoke instruction in B.l(C) that performs a call on the argument. In this case, it makes sense to predict that inlining B.l(C) will result in further inlining of a method invoked on the precise argument.

Listing 3.10 shows a different method of the same object. This method takes an argument of type C but ignores it. This does not make a lot of sense from a design point of view. However, this situation may occur, for example when overriding template methods that provide more arguments than are needed by all implementations.

```
1 class B {
2   void m(C c) {
3     System.out.println(
4       "Argument␣is␣irrelevant.");
5   }
6 }
```

Listing 3.10: Ignoring the argument

When considering A.k() $\overset{?}{\Leftarrow}$ B.m(D $\prec$ C), a query of the dynamic call graph will never yield an outgoing call edge of the form B.$m(C)$ $\rightarrow$ D.m_2. When it is assumed that the call-graph information for B.m(C) is reasonably accurate, it makes sense to conclude that no method is invoked on the argument in this situation.

Listing 3.11 shows the method B.n(C) that takes an object of type C as its argument and invokes a method on another object of the same type.

```
1 class B {
2   void n(C c) {
3     C c2 = ...
4     c2.o();
5   }
6 }
```

Listing 3.11: Invoking a method on a non-argument object

When the optimizing compiler processes the inlining decision A.k() $\overset{?}{\Leftarrow}$ B.n(D $\prec$ C), the dynamic call graph is likely to contain an edge B.n(C) $\rightarrow$ X.n(C) with X :> D. Using only the call-graph data, there is no general way to determine if such an edge is caused by a method invoked on an argument or by a method invoked on another object. Because of this, a heuristic that relies solely on call-graph data to make its predictions will incorrectly predict that a method is invoked on the argument in such a case. To clarify this, the expression "the edge *suggests* that a method is invoked on the argument" is

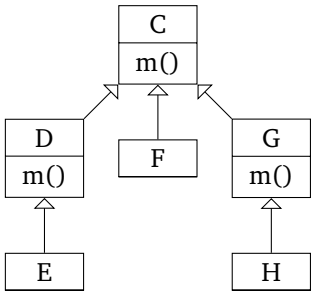

Figure 3.2: A class hierarchy with a virtual method m()

used in the following to signify that the edge may be caused by a method being invoked on the argument, but it is also possible that the method was invoked on a different object.

3.2.1 Matching Edges

In the previously discussed examples, it is plain to see when an edge in the dynamic call graph suggests that a method is invoked on an argument. This is the case when the receiver type of the edge's target method is exactly the same as the precisely known type of the relevant argument.

Because of virtual dispatch, this is not necessarily the case for all combinations of call-edge receiver and argument type. Because the type constraints are different for extant and precise arguments, both cases are examined separately.

Figure 3.2 shows a class hierarchy that is used to illustrate the type constraints for predicting method invocations on extant and precise arguments. The virtual method m() is defined in the root class C. This method is overridden in the classes D, E and G. The classes F and H inherit the definition of m() from their respective supertypes.

Precise-Induced Edges

If a receiver is precise, its type is statically unambiguous. Therefore, when a method is invoked on the precise receiver, there is only a single possible target method for the call site.

Assume that $\mathrm{A.k}(\ldots) \overset{?}{\Leftarrow} \mathrm{B.l}(\ldots, x \prec \mathrm{C}, \ldots)$, with x being any of D, F or H. If the precise argument is of type D, D.m() is the single correct target method when m is invoked on that argument. When inlining a method with such a precise argument, only an edge $\mathrm{B.l}() \rightarrow \mathrm{D.m}()$ suggests a method being invoked on the argument.

If an edge $\mathrm{B.l}() \rightarrow \mathrm{E.m}()$ or $\mathrm{B.l}() \rightarrow \mathrm{C.m}()$ is observed, it can be concluded that the corresponding call was not invoked on the precise argument. It is possible that the observed edge results from B.n(C) being called with a different argument from a different call site in A.k(), or from a different method altogether. In this case, the edge suggests that B.k(C) *does* invoke m() on its argument. At this point, a tradeoff is necessary. If such an edge is counted as suggesting a method invocation on the argument, this may cause more false positives since the type constraints for matching edges are less strict.

Assuming that the local call-graph information for the method B.n(C) is complete, an edge to D.m() will always be present regardless of any other edges with a different receiver type. This means that if the call graph is accurate enough, it is safe to ignore the potentially misleading edge to E.m(). This approach is chosen in the proposed heuristic in order to keep the edge matching simple and statically most correct under the assumption of a locally complete dynamic call graph.

If the argument is of the precise type F, the fact that B.n(C) invokes a method on its argument results in the edge $\mathrm{B.n(C)} \rightarrow \mathrm{C.m}()$. This is the case because C.m() is the valid definition of m() for F. The valid definition is the first definition that is encountered when traversing the superclass hierarchy upwards from F. If the argument is of type H, the valid definition is G.m() for the same reason.

Based on these considerations, a definition for which kinds of edges suggest a call on a precise argument can be formed.

Definition 3.5. *Given a method* $\mathrm{B.n}(\ldots, \mathrm{Y} \prec \mathrm{X}, \ldots)$ *and a complete call graph* $CG = (N, E)$*, if there exists an edge* $e_1 = (\mathrm{B.n}(\ldots, \mathrm{X}, \ldots) \rightarrow \mathrm{Z}.m) \in \mathrm{E}$*, and* $\neg\big(\exists \mathrm{U}.m\big)\big((\mathrm{U} :> \mathrm{Y}) \wedge (\mathrm{U} \ll: \mathrm{Z})\big)$ *holds,* e_1 *is called* precise-induced.

In plain terms, this means that if a method B.n(X) calls a method *m* on its argument of precise type Y, the call graph will show an edge from B.n(X) to the first definition of *m* that is found when traversing the type hierarchy of Y towards its root object. This edge is called *precise-induced*.

Extant-Induced Edges

If the argument is extant instead of precise, the definition has to be modified to accommodate the fact

that the dynamic type of the argument can be any subtype of the static type of the argument.

Assume that the compiler processes $\mathrm{A.k}(\ldots) \overset{?}{\Leftarrow} \mathrm{B.l}(\ldots,\tilde{\mathrm{F}},\ldots)$. Because F has no subclasses, the dynamic receiver type is always F when invoking any method on the argument. When m() is invoked on the extant argument, the call graph will show an edge to C.m() because this is the valid definition of m() for F.

When processing another inlining decision $\mathrm{A.k}(\ldots) \overset{?}{\Leftarrow} \mathrm{B.o}(\ldots,\tilde{\mathrm{D}},\ldots)$, the dynamic receiver of a call invoked on the argument can be of type D or E. Therefore, call-graph edges to D.m() and E.m() can both possibly occur if B.o(D) invokes a method on its argument. When the static type of the argument is C.m(), any of the method definitions C.m(), D.m(), E.m() and G.m() constitute a match. This fact is taken into account by Definition 3.6.

Definition 3.6. *Given a method* $\mathrm{B.n}(\ldots,\tilde{\mathrm{X}},\ldots)$ *and a complete call graph* $CG = (N, E)$*, if there exists an edge* $e_1 = (\mathrm{B.n}(\ldots,\mathrm{X},\ldots) \rightarrow \mathrm{Z}.m) \in E$ *with* $\neg(\exists \mathrm{U}.m)\left((\mathrm{U} :> \mathrm{X}) \wedge (\mathrm{U} \ll: \mathrm{Z})\right)$*,* e_1 *is called* extant-induced*. If there exists an edge* $e_2 = (\mathrm{B.n(X)} \rightarrow \mathrm{Y}.m) \in E$ *with* $\mathrm{Y} <: \mathrm{X}$*,* e_2 *is also called* extant-induced*.*

Definition 3.6 is similar to Definition 3.5, but uses the static type of the argument instead of the precise type to define the type constraints for the call-graph edge. An invoke instruction in B.n(X) may manifest itself in the call graph in the same way that it does in Definition 3.5. It may also be shown by an edge to a subtype implementation of the method in question. Note that any combination of these edges may occur: both, none, or only one of the two. The kind of edge that is actually observed in the call graph depends on the dynamic types the receiver actually has when the method is executed.

Definition 3.5 and Definition 3.6 form the proposed heuristic. The definitions describe what kind of call-graph edges are caused by invoking a method on a precise or extant argument. In order to predict further inlining on these arguments, the reverse of the implication is required. The proposed heuristic inspects the edges that are present in the dynamic call graph to determine whether it is likely that a method is called on an argument. This approximation is wrong if the method is invoked on a non-argument object or the dynamic call graph does not contain the induced edge because it was not sampled. In all other cases, the heuristic is correct.

3.2.2 Special Cases

There are two cases in which Definition 3.5 and Definition 3.6 predict further inlining when it is possible to statically determine that either no further inlining will occur or no indirect benefit is likely. These cases are:

- further inlining of non-virtual calls; and
- further inlining at optimization level 0.

The actual indirect benefit that the heuristic is used to predict only materializes if guards can be omitted when further inlining is performed. Because of this, it is necessary to only predict an indirect benefit if the observed call edge represents a call that is *dispatched virtually*. In cases where no virtual dispatch takes place, the target method to inline is unambiguous, and no guards are necessary in the first place.

Suppose that when processing the inlining decision $\mathrm{A.m}() \overset{?}{\Leftarrow} \mathrm{B.n}(\mathrm{D} \prec \mathrm{C})$, a call-graph edge is observed that matches the criteria in Definition 3.5. The dynamic call graph provided by the Jikes RVM profiling infrastructure contains call site information consisting of the containing method and the bytecode index of the call site. This allows the heuristic to inspect the bytecode of the call site in question. If the call site in question is a non-virtual **invokespecial** or **invokestatic** instruction, guardless inlining of this call site is possible without requiring an extant or precise receiver. In this case, the heuristic does not interpret the edge corresponding to the non-virtual call site as a match.

The second special case is related to the inlining behavior of the optimizing compiler. In Jikes RVM, optimization level 0 is chosen for most methods compiled with the optimizing compiler. At this optimization level, only trivial inlining is performed. Inlining methods that are bigger than a predefined size limit is always rejected. This means that a large number of call edges that suggest further inlining are never actually inlined.

To remedy this situation, the size of the target method of the edge that suggests further inlining is compared to the maximum size for trivial methods, adjusted for the maximum size-estimate bonus that can be assigned. This allows methods that are so large that they can never possibly be inlined at optimization level 0 to be rejected by the heuristic. In Jikes RVM, the compiled size of a method is estimated when the method is loaded. The maximum size reduction that can be granted to a method is a fixed at 40% of the total size. Using this information, the heuristic can estimate if a target method can ever become small enough to be inlined at optimization level 0.

3.2.3 Modifying the Call-Graph Profiler

The dynamic call-graph profile collected by Jikes RVM can only be queried for outgoing edges

on a per-call-site basis instead of per method. When computing an inlined-size estimate, only the method for which to estimate the size is known. Because of this, the proposed heuristic needs a facility that allows querying the sampled call sites for arbitrary methods.

To solve this problem, the call-graph sampling mechanism for Jikes RVM was modified. Whenever a call edge is sampled, the call site is added to a set of sampled call sites for the method. This set can then be queried for each method. This allows the proposed heuristic to determine all outgoing edges for all call sites of a given inlining candidate.

3.3 Modifying the Size-Estimation Heuristic

The proposed heuristic yields a per-argument prediction that states if propagating the precise type or extant state of this argument into the callee context is likely to make one or more guard tests in the callee unnecessary. To make use of this prediction, it must be integrated with the general inlining heuristic of Jikes RVM (cf. Section 2.4.2).

When deciding an inlining problem $A.m(\ldots) \overset{?}{\Leftarrow} B.n(\ldots, D \prec C, \ldots)$, the inline oracle of Jikes RVM reduces the size estimate of B.n(C) by 15%. Similarly, when deciding $A.m(\ldots) \overset{?}{\Leftarrow} B.n(\ldots, \tilde{C}, \ldots)$, the size estimate is reduced by 5%. These bonuses are granted for each precise or extant argument present at the call site. The rationale for the bonuses is that the propagated argument information enables guardless inlining of calls invoked on the argument. A precise argument additionally allows type checks to be eliminated. An extant argument allows guardless inlining only if the call site is proved *currently monomorphic*. When this assumption is invalidated by future class loading, it becomes necessary to recompile the caller method. Due to those differences in the induced benefit, the extant argument bonus is lower than the bonus for precisely typed arguments.

Currently, this bonus is always attributed when inlining a method with a number of precise or extant arguments is considered. In the proposed solution, the bonus is only attributed if an edge in the call graph suggests that further inlining of a method invoked on the argument is likely to occur, according to the criteria in Definition 3.5 and Definition 3.6.

There is one problem with this approach. The bonus is either attributed fully or not at all based on the prediction of the proposed heuristic. This means that the elimination of type checks is no longer incorporated into the size estimate. Handling this correctly would require splitting the bonus for precise arguments into a static part that is always attributed and a dynamic part that is attributed based on the prediction of the heuristic. To determine the split between the bonuses, extensive benchmark-based tuning is required. Instead, this limitation is accepted for now and reserved as a future opportunity for optimization.

3.4 Solution Variants

In Section 3.3, a straightforward approach of incorporating profile-based predictions of further inlining into an inlining heuristic based on size estimates was presented. This is the approach suggested by the structure of the unmodified inline oracle. Reasoning about call hierarchies in object-oriented programs leads to an incremental refinement of the heuristic. As a more invasive alternative, an approach that aims to exploit indirect benefits more often is suggested.

3.4.1 Deep Inlining

All discussion of further inlining so far was restricted to a single additional level of inlining. Recall the notation for further inlining in the presence of a precise argument:

$$A.m(\ldots) \Leftarrow B.n(\ldots, Y \prec X, \ldots) \Leftarrow Y.u(\ldots)$$

In this example, the precise type information is propagated over a single call boundary. Listing 3.12 shows an example where a benefit is achieved after propagating the information over two call boundaries.

```
1  class A {
2    void m() {
3      B b = ...
4      X x = new Y();
5      b.n(x);
6    }
7  }
8
9  class B {
10   void n(X x) {
11     C c = ...
12     c.o(x);
13   }
14 }
15
16 class C {
17   void o(X x) {
18     x.u()
19   }
20 }
```

Listing 3.12: Further inlining after two steps

If all calls are inlined, this results in an inlining sequence in which the precise type information is

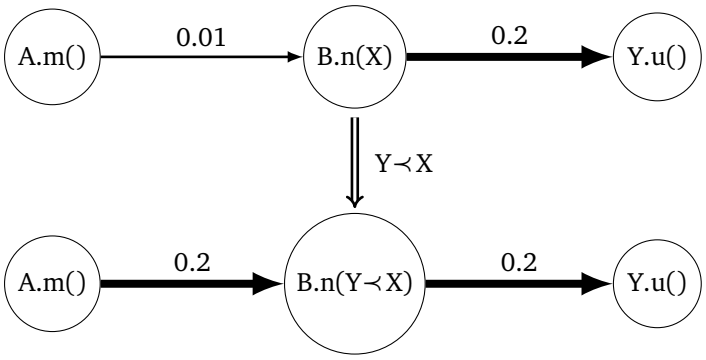

Figure 3.3: Conductive call-graph edges

propagated over two method boundaries before it enables guardless inlining of Y.u().

$$\begin{aligned} \text{A.m}(\ldots) &\Leftarrow\!\!\shortmid \text{B.n}(\ldots, \text{Y} \prec \text{X}, \ldots) \\ &\Leftarrow\!\!\shortmid \text{C.o}(\ldots, \text{Y} \prec \text{X}, \ldots) \Leftarrow\!\!\shortmid \text{Y.u}(\ldots) \end{aligned} \tag{3.8}$$

Here, it is appropriate to grant B.n(X) a size reduction bonus because inlining it is likely to result in guardless inlining of Y.u() if C.o(X) is inlined into the root method as well. However, the proposed heuristic does not respect such cases. Only B.n(X) is queried for call edges that suggest that a method is invoked on the precise argument.

Because of this, the proposed heuristic is modified to address this problem for inlining sequences of arbitrary length. In the example, the fact that B.n(X) calls C.o(X) is likely to result in a call-graph edge that can be found by the modified heuristic. Call-graph nodes allow accessing the complete method descriptor that includes the types of the formal arguments.

The proposed heuristic is modified so that the possible flow of a precise or extant argument along the call graph is evaluated. When the compiler decides the inlining problem $\text{A.m}(\ldots) \overset{?}{\Leftarrow\!\!\shortmid} \text{B.n}(\ldots, \text{Y} \prec \text{X}, \ldots)$, the call graph of B.n(X) is queried for edges. When a precise-induced edge is encountered, the heuristic predicts further inlining. If an edge to a method that declares an argument that may contain the precise argument is encountered, the heuristic recurses by processing the callee of this edge in the same way as B.n(X). A method to which the precise argument can be propagated has the form $\text{C.o}(\ldots, \text{Z}, \ldots)$, where $\text{Y} <: \text{Z}$. When deciding the inlining problem $\text{A.m}(\ldots) \overset{?}{\Leftarrow\!\!\shortmid} \text{B.n}(\ldots, \tilde{\text{X}}, \ldots)$, the call graph is queried for extant-induced edges. If a different edge is observed that has the form $\text{C.o}(\ldots, \text{Z}, \ldots)$, where $(\text{X} <: \text{Z}) \vee (\text{Z} <: \text{X})$, the callee is treated in the same way as the original callee of the inlining problem, causing the heuristic to recurse. This algorithm terminates a branch of depth-first search when the encountered call-graph node was already visited or does not declare a matching argument. In practice, it has been determined that this is a sufficiently effective termination condition and no restriction of the recursion depth is necessary.

3.4.2 Making Call-Graph Edges Heat-Conductive

The original call-graph-based heuristic and the improvement for deep inlining presented in Section 3.4.1 work by removing the argument bonus if it is unlikely that the callee will invoke a method on that argument. Because of this, it is to be expected that the modified heuristic inlines fewer methods than the original heuristic.

In order to fully exploit the potential for intraprocedural optimizations enabled by propagating precise and extant arguments, it may be desirable to encourage inlining of methods for which additional indirect benefits are expected. One straightforward approach to do so is to increase the bonuses that are attributed per argument. This should, in theory, cause methods to be inlined that would never have been inlined by the unmodified inlining heuristic. However, this approach is unsatisfying because it relies on the tuning of parameters instead of an evident mathematical modeling of the cost-benefit tradeoff.

When the inline oracle of Jikes RVM that was described in Section 2.4.2 decides whether to inline a particular call edge, that edge's frequency is factored into the cost-benefit tradeoff. The *edge temperature* that represents the call frequency is used to favor particularly hot edges, because the potential payoff is highest for inlining calls that are frequently executed.

Applying this principle to further inlining suggests the concept of *conductive call-graph edges*. Figure 3.3 shows an example call graph with further inlining. The temperature of the edge from A.m() to B.n(X) is low at 0.01, while the edge from B.n(X) to Y.u() has a higher temperature of 0.2. If A.m() is the only caller of B.n(X), this may be caused by B.n(X) calling Y.u() 20 times inside a loop.

When deciding the inlining problem $\text{A.m}() \overset{?}{\Leftarrow\!\!\shortmid} \text{B.n}(\text{Y} \prec \text{X})$ on the basis of the call graph in Fig-

ure 3.3, the heuristic defined in Section 3.2 will correctly predict further inlining for the precise argument. However, depending on the chosen inlining thresholds, the edge from A.m() to B.n() may not be hot enough to warrant inlining. At the same time, the higher temperature of the edge from B.n(X) to Y.u() may cause this edge to be inlined.

This leads to a suboptimal situation where the second, hot edge may need to be inlined with a guard test because the type information was not propagated by inlining the first, colder edge. This is problematic because the guard test needs to be executed 20 times for each call of B.n(X).

Because the proposed heuristic provides the information that guardless inlining of the hot edge is likely, this problem can be fixed by setting the temperature of the cold edge to the temperature of the hot edge as shown in the lower part of Figure 3.3. The metaphor here is that the call graph as seen by the inline oracle becomes *thermally conductive* in the presence of precise or extant arguments.

The modified heuristic works by replacing the temperature of any inlining edge by the maximum of its own temperature and the temperature of all edges that suggest an indirect benefit due to precise or extant arguments as defined by Definition 3.5 and Definition 3.6. This causes the inline oracle to favor cold edges that may induce an indirect benefit for further inlining of call edges that are executed often.

4 Evaluation

In this chapter, the presented problem and the proposed solution are evaluated quantitatively using a benchmark suite. The general prevalence of precise and extant arguments is examined. Then, a quality metric for heuristics that predict further inlining is presented and applied to determine the prediction quality of the proposed heuristic. In a last step, the performance of all proposed heuristics is compared to the performance of the unmodified heuristic in terms of pure program execution time, time spent compiling, and total running time.

This chapter addresses a number of questions regarding the presence of precise and extant arguments and their consequences for inlining heuristics. In order to acquire an understanding of the problem space, the following questions are of interest:

- How frequently does the compiler encounter precise and extant arguments and receivers?
- How does assigning bonuses for precise and extant arguments influence performance?

In Chapter 3, a heuristic was presented that assigns bonuses for precise and extant arguments in a way that is different from the original heuristic. Two modified variants of the proposed heuristic were also presented. The deep-inlining variant recursively traverses the call graph to find precise-induced and extant-induced edges. The conductive-call-graph variant encourages inlining when an extant or precise benefit is expected for inlining a hot edge. To evaluate if the presented heuristics constitute an improvement over the unmodified heuristic, the following questions are addressed:

- What is the per-decision quality of the proposed heuristic?
- How does the per-decision quality of the proposed heuristic compare to the unmodified heuristic?
- Are the decisions made by the proposed heuristic better than deciding randomly?
- What is the proposed heuristic's effect on program time, compile time and aggregate performance?
- What is the performance effect of the variants of the proposed heuristics?

Section 4.1 describes the evaluation methods used to answer these questions. In Section 4.2, the frequency of precise and extant arguments and receivers is examined. In Section 4.3, a method for determining the quality of further inlining predictions is described and applied to the proposed heuristic. In Section 4.4 to Section 4.8, performance measurements of different heuristics for attributing precise and extant argument bonuses are presented. Section 4.9 concludes the performance evaluation by providing a summary of the results of the performance measurements.

4.1 Evaluation Methods

In order to answer general questions about precise and extant benefits as well as the performance impact of the presented heuristics, two main evaluation techniques are used:

- Untimed benchmarking runs in which the individual decisions of the inlining oracle are recorded provide insight into how the process of making inlining decisions is affected by the proposed heuristics.
- Timed benchmarking runs evaluate the aggregate effect of the presented heuristics on performance.

For both kinds of setup, the Dacapo 2006-10-MR2 benchmark suite is used. Also, most measurements are performed with the help of replay compilation to isolate the effects of the presented compiler modifications while reducing the influence of non-determinism caused by other subsystems. The following subsections describe the individual benchmarks of the Dacapo suite and explain the measures that were taken to reduce the influence of non-determinism on the performance measurements. In addition to this, Section 4.1.4 describes the approach that was used to simulate longer-running computations.

4.1.1 Benchmarks

The client program that is executed by the virtual machine and thus subject to compilation is simulated by the benchmarks of the Dacapo 2006-10-MR2 benchmark suite [Bla+06]. The individual benchmarks simulate the following kinds of application:

antlr generates parsers and lexical analyzers.

bloat performs bytecode analysis and optimization.

chart plots graphs and renders them as PDF.

eclipse runs performance tests for the Java development tools that are part of the Eclipse IDE.

fop generates a PDF file from an XSL-FO file.

hsqldb simulates a banking application running on an in-memory HSQLDB database.

jython uses the Jython implementation of Python to execute the pybench benchmark.

luindex creates a search index from literary documents.

lusearch performs keyword searches on a search index of literary documents.

pmd analyzes Java source code for problems.

xalan uses XSLT to transform XML documents.

Of the 11 available benchmarks, only 10 are used for the measurements that are based on replay compilation (cf. Section 4.1.2). The benchmark eclipse is omitted because it cannot be used in conjunction with repay compilation as supported by Jikes RVM.

One of the design goals for the Dacapo benchmark suite was to use widely-used open-source Java programs as opposed to creating synthetic benchmarks. Other benchmarks usually focus on processor-intensive numeric computations, such as the benchmarks compress, mpegaudio and mtrt in SPECjvm98 [Spe]. These benchmarks simulate purely computational loads that are implemented using loop constructs. In contrast, the Dacapo approach leads to benchmark code that has an object-oriented, interprocedural control flow [Bla+08]. Because the proposed heuristic addresses the domain of interprocedural optimization, the Dacapo benchmark suite is a good match for assessing its quality.

Table 4.1 shows an overview of the performance characteristics of the selected benchmarks.

4.1.2 Controlling Non-Determinism

Benchmarking a managed execution environment typically yields large variations in the reported results. Frequency and timing of garbage collections are different for every run. Since Jikes RVM uses the same heap for application objects and VM objects, compilation may also trigger garbage collections. Garbage collections can have an influence on execution time as well as the measured compile time, which is undesirable when evaluating compiler modifications. Because of this, a large maximum heap size of 512 MiB was used for all measurement runs in order to reduce the number of major garbage collections.

In addition to the garbage collector, there are other subsystems that cause non-determinism. Jikes RVM's adaptive optimization system [Arn+00b] uses live profile data to decide when particular methods should be recompiled using the optimizing compiler. Along with the decision to recompile, the system also decides at which optimization level the method should be recompiled. These decisions vary between different runs of the same benchmark. This is problematic because it also affects the performance of the compiler itself and that of the compiled code. Inlining is restricted to guardless inlining of small methods when compiling at optimization level 0. At optimization levels 1 and 2, more aggressive inlining is performed. This means that the decisions of the adaptive optimization system have a large influence on which call sites are inlined and may also cause large variations in compile time and running time of the compiled code. This makes it hard to evaluate the effect of specific compiler modifications.

Replay Compilation

To improve the situation, Georges, Eeckhout, and Buytaert [GEB08] recommend the use of *replay compilation*. The process involves generating a set of *compilation plans* by performing multiple *profile runs* of each benchmark. A compilation plan consists of three types of recorded profile information:

- an *advice file* that specifies the compiler and optimization level to compile each method with;
- an *edge-counter profile* that is used to determine the execution frequency of the basic blocks in method control flow graphs; and
- a *call-graph profile* that contains weighted call edges for the executed program.

The generated compilation plans are then used in the timed *replay runs*. Whenever a method is first invoked in a replay run, it is compiled with the compiler and optimization level specified in the advice file. The pre-recorded edge counter profile is used for control flow optimizations such as code reordering. The call-graph profile is used for profile-based decisions of the compiler, such as determining which call edges to inline. Using replay compilation results in a consistent compilation load for each invocation of a particular compilation plan. This eliminates most of the non-deterministic effects caused by the adaptive optimization system. Therefore, the technique of replay compilation is a useful tool for comparing different compiler versions [GEB08].

		Time spent compiling (%)			Code generated (KiB)	
Benchmark	Running time (ms)	Baseline	Opt	Combined	Baseline	Opt
antlr	2,709↓	7.59↑	16.70	24.29↑	1,040.4	53.1
bloat	11,567	1.48	19.76	21.24	1,022.8	140.9
chart	12,092	1.38	10.35	11.74	1,200.3	92.2
fop	2,828	5.72	6.21↓	11.92	1,298.4	21.3↓
hsqldb	4,412	2.63	20.56↑	23.19	709.6	80.2
jython	9,171	3.81	8.47	12.28	2,651.1↑	92.0
luindex	11,414	0.68↓	10.28	10.95↓	367.2	116.2
lusearch	10,930	2.10	13.30	15.41	256.6↓	124.0
pmd	8,733	2.07	11.60	13.67	1,018.7	86.4
xalan	14,843↑	1.09	13.64	14.72	846.7	258.8↑

↓ = lowest,↑ = highest

Table 4.1: The performance characteristics of the selected benchmarks

Running two versions of the virtual machine with the same compilation plan yields a *matched pair* of performance measurements. Differences in running time, code size and compile time are very likely to be caused by the differences in the compilers and not by decisions made in other subsystems of the virtual machine. Comparing matched pairs of performance measurements generally yields more significant results than comparing the same number of unmatched measurements [GEB08].

The proposed heuristic relies on profiled call-graph data to make its predictions. A major drawback of using replay compilation is the fact that the call-graph data from the profile run is completely loaded at startup. This means that when the proposed heuristic makes predictions, all call edges that were sampled until the end of the profile run are available. In a non-replay run, the call graph is constructed gradually over the course of the run. This may yield an advantage to the proposed call-graph-based heuristic that does not occur in a real-world setting.

Another problem with replay compilation is that every method is compiled at the optimization level specified in the advice file when it is first executed. In a real-world setting, a method may be recompiled a number of times before reaching its final optimization level. This means that differences in compile time that are observed when replay compilation is used are not necessarily representative of the compile-time differences observed in a non-replay setting.

To solve both problems, separate benchmarking runs without replay compilation are performed to confirm the initial results (cf. Section 4.8).

4.1.3 Measured Quantities

When performing timed benchmarking runs, the following quantities are measured:

- the running time for each iteration;
- the time spent compiling methods with the optimizing compiler;
- the time spent compiling methods with the baseline compiler;
- the amount of machine code generated by the optimizing compiler; and
- the amount of machine code generated by the baseline compiler.

Time is measured in milliseconds (ms) and the amount of generated code is measured in kibibytes (KiB). Subtracting the compile time of both compilers from the total running time yields the pure *program time*. This is the amount of time that the compiled code of the program and the VM needs to perform all of its operations, including the overhead caused by the garbage collector and other non-compiler parts of the VM. Program time is a suitable quantity to measure *code quality*. The less time the compiled code needs to do the actual work, the better quality the code is.

Table 4.1 shows some performance metrics of the unmodified heuristic for the selected benchmarks. The results were obtained by executing each benchmark using ten different compilation plans. The numbers shown for each benchmark represent the arithmetic mean computed over the results for all ten compilation plans. All benchmarks were run in single-iteration mode. The measurements were performed on a computer with an AMD Athlon 64 3200+ processor (2.2 GHz clock speed, 128 KiB L1 cache total, 512 KiB L2 cache) and 1024 MiB RAM running the 32-bit version of Linux (kernel version 2.6.32) in single-user mode. The *production* configuration of Jikes RVM was used. In this configuration, most VM code is precompiled at optimization level 2. All results that are presented in this thesis were measured with the same machine and configuration.

The second column of Table 4.1 shows the total running time of the benchmarks. Antlr is the shortest-running benchmark at an average running time of about 2,700 milliseconds, while xalan has the longest total running time at about 14,800 milliseconds average. The next three columns show the percentage of total running time that the virtual machine spends compiling code. This includes the code of the benchmark, the code of the Java Runtime Environment (JRE), and the VM code that was not precompiled. The first of these columns shows the percentage of total running time spent in the baseline compiler, while the second column shows how much time is spent compiling code with the optimizing compiler. The baseline compile-time ratio varies from 0.68 % for luindex to 7.59 % for antlr. Time spent in the optimizing compiler ranges from 6.21 % for fop to 20.56 % for hsqldb. The ratio of time spent in the optimizing compiler to time spent in the baseline compiler varies very strongly. While antlr spends about a third of its compile time in the baseline compiler, bloat spends over 90 % of its compile time in the optimizing compiler. Overall, compile time accounts for 10.95 % of total running time for luindex up to 24.29 % for antlr. The last two columns show the amount of machine code that is generated by each of the compilers. The size of the code generated by the baseline compiler ranges from 256.6 kB for lusearch to 2,651.1 kB for jython. On the other hand, the optimizing compiler produces much less code, with a minimum of 21.3 kB for fop and a maximum of 258.8 kB for xalan. The fact that the optimizing compiler is only responsible for a small fraction of the total code-size footprint suggests that reducing the size of the code generated by the optimizing compiler without also improving the execution speed of the code is not a top performance concern.

4.1.4 Long-Running Loads

As Table 4.1 shows, the average execution time of the selected Dacapo benchmarks is between 2 and 15 seconds. This means that the benchmarks simulate programs that terminate very quickly. There are applications that run for hours or even days on a Java VM. These include interactive Desktop applications as well as Server applications. Therefore, it is inappropriate to only use quickly-terminating programs to measure the performance of a virtual machine. A longer-running program leads to more methods being compiled using the optimizing compiler and to higher overall optimization levels. Because of this, it is useful to also simulate long-running loads when evaluating compiler modifications.

For this, the Dacapo benchmark suite provides sets of input data in the sizes *small*, *default*, and *large* that, in theory, should lead to different running times for the benchmarks. The running time range quoted above is based on the default input-data size. It is possible to increase the total running time by using the large input-data size instead. However, this is problematic because the relative difference in running time between default and large is different for every benchmark. For some benchmarks, no large set of input data is provided at all. Because of this, varying the size of the input data seems unsuitable in order to achieve consistent scaling of the computational load.

Therefore, a different approach is used here. To increase the load, multiple iterations of each benchmark are executed using the Dacapo benchmark harness. This allows running time to be scaled up for each of the selected benchmarks. While the first iterations are slower because more code needs to be compiled and the optimization levels are lower, later iterations become progressively faster.

When this method is used, it is necessary to select the measured quantities carefully. The Dacapo benchmark harness yields separate running times for each iteration of the benchmark. These times do not include the time that was needed to verify the result of the computation performed by the benchmark. Compilation time is reported when the virtual machine terminates. Adding the individual iteration times yields a total time. When compile time is subtracted from this, the total program time for performing the set number of iterations of the benchmark can be computed.

Note that increasing the number of iterations and totaling the individual times does not represent a realistic load: The same problem is computed over and over again as opposed to computing different problems. Therefore, the primary setup used in the evaluation of the proposed heuristics is the normal single-iteration case. The results of simulating long-running loads in a setting without replay compilation are included in Appendix A.

4.2 Prevalence of Precise and Extant Arguments

The heuristics discussed in this thesis are concerned with the influence of precise and extant arguments on the cost-benefit tradeoff of inlining. To determine how relevant these arguments are, it is necessary to quantify how often they are actually encountered by the compiler.

4.2.1 Setup

For each of the selected benchmarks of the Dacapo suite, ten compilation plans were run with a modified optimizing compiler. This compiler was instru-

	Non-receiver arguments (%)		Receivers (%)		All arguments (%)	
Benchmark	Extant	Precise	Extant	Precise	Extant	Precise
antlr	19.44	37.95	27.12	35.39	25.14	35.97
bloat	19.40	44.34	7.89↓	69.08↑	12.59↓	58.97↑
chart	12.37	39.71	12.97	58.36	12.72	50.46
fop	47.52↑	21.04	38.67	30.79	43.86	25.09
hsqldb	21.83	13.38	15.64	29.53	18.82	21.18
jython	43.95	15.38	27.89	47.09	37.25	28.63
luindex	7.03↓	60.84↑	39.46	21.28↓	32.94	29.23
lusearch	22.11	41.93	59.82↑	23.36	51.10↑	27.66
pmd	12.29	26.67	17.89	43.24	15.27	35.49
xalan	38.05	11.34↓	27.59	24.80	31.91	19.25↓
Arithmetic mean	24.40	31.26	27.50	38.29	28.16	33.19

↓ = lowest,↑ = highest

Table 4.2: Prevalence of the precise and extant properties of object-reference arguments in the Dacapo benchmark suite. The first two numeric columns show the percentage of non-receiver object-reference arguments that are extant or precise. The next two numeric columns show the percentage of extant or precise receivers. In the last two columns, the percentage of extant or precise arguments in all arguments, including receivers, is shown.

mented to output a number of statistics for each compiled call site:

- whether the call site's receiver is extant, precise or none of the former;
- the number of precise arguments;
- the number of extant arguments; and
- the number of object-reference arguments that are neither precise nor extant.

Arguments and receivers that are both extant and precise count as precise only since this is the stronger criterion. Note that only call sites compiled by the optimizing compiler are included in the statistic. Inlining and especially the assignment of size bonuses for precise or extant arguments only takes place in the optimizing compiler. Because of this, restricting the set of analyzed method-call instructions to those compiled by the optimizing compiler means that only those arguments and receivers are counted for which the proposed heuristic is relevant.

Only call sites in benchmark code and JRE code were analyzed for their arguments. Since the production configuration of Jikes RVM was used, most VM code is precompiled. The VM code that is compiled at runtime consists mostly of runtime-service methods that perform functions like object allocation. Inlining of these methods is controlled by annotations, which means that inline decisions for those call sites are formed without making a cost-benefit tradeoff at runtime. Because of this, call sites in Jikes RVM methods were discarded. Using the measured argument counts, summary percentages were computed on a per-plan basis. From this, the per-benchmark arithmetic mean and a summary arithmetic mean over all benchmarks were computed.

4.2.2 Results

Table 4.2 shows the result of the measurement normalized to object-reference arguments and receivers, respectively. The first two numeric columns show the percentage of extant and precise non-receiver object-reference arguments. Primitive arguments and array arguments are not counted into the statistic. The implicit **this** arguments that are passed to **invokevirtual**, **invokeinterface** and **invokespecial** instructions are also not part of the statistic shown in the first two columns. The percentage of extant arguments ranges from 7.03 % for luindex to 47.52 % for fop. On average, 24.4 % of object-reference arguments are extant but not precise. The percentage of precise arguments ranges from 11.34 % for xalan to 60.84 % for luindex. On average, 31.26 % of non-receiver object arguments are precise. This means that overall, over 50 % of all non-receiver object-reference arguments are either precise or extant and are assigned a bonus by the unmodified heuristic.

The two middle columns show the percentage of extant and precise receivers, respectively. The benchmark bloat has the lowest percentage of extant receivers at 7.89 %. The percentage is highest for lusearch at 59.82 %. Precise receivers are least common in luindex (21.28 %) and occur most often in bloat (69.08 %). On average, 27.5 % of receivers are extant and 38.29 % of receivers are precise. This means that more than half of all non-

static call sites can be guardlessly inlined without the need for on-stack replacement.

The last two columns show the combined statistic for all object arguments including the receivers. The target method of a call site may invoke another method on its receiver via the **this** argument. If **this** is extant or precise, the same kind of benefit materializes as when invoking a method on another argument. Therefore, receiver arguments also need to be included in the statistic. Overall, the percentage of extant arguments ranges from 12.59 % for bloat to 51.1 % for lusearch. The overall percentage of precise arguments ranges from 19.25 % for xalan to 58.97 % for bloat. The average prevalence of extant arguments is 28.16 % and 33.19 % for precise arguments. These averages, as well as the minimum and maximum values, are slightly higher than the percentages that do not include receivers.

Benchmark	Call sites with precise or extant arguments (%)
antlr	56.40
bloat	55.87
chart	48.68
fop	76.89↑
hsqldb	46.95
jython	63.75
luindex	42.65↓
lusearch	73.95
pmd	56.91
xalan	52.11
Arithmetic mean	57.42

↓ = lowest,↑ = highest

Table 4.3: The percentage of call sites that have at least one precise or extant argument (including receivers)

Table 4.3 shows the percentage of call sites compiled with the optimizing compiler that have at least one extant or precise argument. Call sites that only have an extant or precise receiver are included in this statistic. The percentage is lowest for luindex at 42.65 % and highest for fop at 76.89 %. On average, 57.42 % of call sites have at least one precise or extant argument. This means that the proposed heuristic affects the inlining decisions for 57.42 % of call sites.

The data presented here shows that extant and precise arguments are encountered in more than half of all inlining decisions. Furthermore, about half of all object-reference arguments of call sites are extant or precise. It can be concluded that taking account of such arguments is integral to making a correct cost-benefit tradeoff when deciding whether to inline a particular call site.

4.3 Prediction Accuracy

The proposed heuristic that was described in Section 3.2 uses Definition 3.5 and Definition 3.6 to predict when it is likely that a callee method invokes another method on one of its extant or precise arguments. If this is the case, an indirect benefit is likely to occur if the method in question is inlined. To determine the quality of the proposed heuristic, the predictions made by the heuristic must be compared to the actual inlining decisions of the compiler. Definition 4.1 describes the predictions made by the proposed heuristic.

Definition 4.1. *Given a set X of extant arguments in a program, the set E defined in Equation 4.1 contains an instance of the corresponding inlining problem for each extant argument.*

$$E = \left\{ T_i.m_i(\ldots) \overset{?}{\Leftarrow} U_i.n_i(\ldots, \tilde{x}_i, \ldots) \mid x_i \in X \right\} \tag{4.1}$$

A function $\mathscr{H}_{\text{extant}} : E \to \{\top, \bot\}$ is defined that evaluates to the prediction made by the proposed heuristic based on Definition 3.6.

Given a set Y of static argument types and a set Z of precise types in a program, the set P defined in Equation 4.2 contains the corresponding inlining problem for each precise argument.

$$P = \left\{ T_i.m_i(\ldots) \overset{?}{\Leftarrow} U_i.n_i(\ldots, z_i \prec y_i, \ldots) \mid z_i \in Z \wedge y_i \in Y \right\} \tag{4.2}$$

A function $\mathscr{H}_{\text{precise}} : P \to \{\top, \bot\}$ is defined that evaluates to the prediction made by the proposed heuristic based on Definition 3.5.

$\mathscr{H}_{\text{precise}}$ and $\mathscr{H}_{\text{extant}}$ as well as $\mathscr{P}_{\text{precise}}$ and $\mathscr{P}_{\text{extant}}$ (cf. Definition 3.2 and Definition 3.3) can be computed after the executed program has terminated. The result of both predicates $\mathscr{P}_{\text{precise}}$ and $\mathscr{P}_{\text{extant}}$ is false if no further inlining of a method takes place that has a receiver type that is similar to the given extant or precise argument. For the purpose of the accuracy evaluation, it makes sense to distinguish between the case where the prediction is irrelevant because the initial inlining problem was not inlined, and the case where the initial problem was inlined but no relevant further inlining occurred. The first case cannot be included in the accuracy statistic because it is impossible to determine the result of $\mathscr{P}_{\text{precise}}$ and $\mathscr{P}_{\text{extant}}$ if the relevant problem was not inlined. Because of this, the sets $E' \subset E$ and $P' \subset P$ are defined that only include those inlining problems for which a positive inlining decision was reached.

The problem of identifying arguments for which further inlining is likely to occur can be interpreted as an information-retrieval problem. An information-retrieval system presents the user with

	Observed behavior	
Predicted behavior	Further inlining	No further inlining
Further inlining	True positive (t_p)	False positive (f_p)
No further inlining	False negative (f_n)	True negative (t_n)

Table 4.4: A binary classification for comparing the prediction of the proposed heuristic to the actual inlining behavior of the compiler

a set of documents matching a specific information need. The goal of the system is to select as many relevant documents as possible, and as few irrelevant documents as possible out of the set of all documents. With regard to the problem of predicting further inlining on extant and precise arguments, the extant or precise arguments constitute the set of all documents. The arguments for which further inlining occurs is the set of relevant documents and the arguments for which the proposed heuristic predicts further inlining is the set of retrieved documents.

Manning, Raghavan, and Schütze [MRS08, pp. 142-144] describe how the quality of the results of an information retrieval system can be quantified using the measures *accuracy*, *precision* and *recall*. These measures are computed from the cases of a binary classification. Comparing the prediction made by the proposed heuristic to the actually-observed outcome leads to the binary classification shown in Table 4.4.

In the *true positive* case, the heuristic predicts further inlining that subsequently occurs. In the *true negative* case, the heuristic correctly predicts that no further inlining occurs. In the *false negative* case, the heuristic predicts that no further inlining is likely, but further inlining is actually observed. In the *false positive* case, the heuristic predicts further inlining, but none occurs.

The number of predictions in the classes t_p, f_p, f_n and t_n can be computed from the benefit predicates and the heuristic functions. The Set of Equations 4.3 shows how the classification can be computed for the prediction of further inlining on extant arguments.

$$\begin{aligned} t_p &= \left|\{e \in E' \mid \quad \mathscr{P}_{\text{extant}}(e) \wedge \quad \mathscr{H}_{\text{extant}}(e)\}\right| \\ f_p &= \left|\{e \in E' \mid \quad \mathscr{P}_{\text{extant}}(e) \wedge \neg\mathscr{H}_{\text{extant}}(e)\}\right| \\ f_n &= \left|\{e \in E' \mid \neg\mathscr{P}_{\text{extant}}(e) \wedge \quad \mathscr{H}_{\text{extant}}(e)\}\right| \\ t_n &= \left|\{e \in E' \mid \neg\mathscr{P}_{\text{extant}}(e) \wedge \neg\mathscr{H}_{\text{extant}}(e)\}\right| \end{aligned} \tag{4.3}$$

The Set of Equations 4.4 shows how the classification can be computed for precise arguments.

$$\begin{aligned} t_p &= \left|\{p \in P' \mid \quad \mathscr{P}_{\text{precise}}(p) \wedge \quad \mathscr{H}_{\text{precise}}(p)\}\right| \\ f_p &= \left|\{p \in P' \mid \quad \mathscr{P}_{\text{precise}}(p) \wedge \neg\mathscr{H}_{\text{precise}}(p)\}\right| \\ f_n &= \left|\{p \in P' \mid \neg\mathscr{P}_{\text{precise}}(p) \wedge \quad \mathscr{H}_{\text{precise}}(p)\}\right| \\ t_n &= \left|\{p \in P' \mid \neg\mathscr{P}_{\text{precise}}(p) \wedge \neg\mathscr{H}_{\text{precise}}(p)\}\right| \end{aligned} \tag{4.4}$$

Counting the cases of the binary classification allows *accuracy*, *precision* and *recall* [MRS08, pp. 142-144] to be computed for the heuristic. Accuracy is the ratio of correct predictions to the number of all predictions for which an actual result was observed.

$$a = \frac{t_p + t_n}{t_p + f_p + t_n + f_n} \tag{4.5}$$

For the proposed heuristic, accuracy is a suitable metric for the general rate of mispredictions. The rate of mispredictions is $1 - a$.

Precision measures how often a prediction of further inlining is correct as opposed to yielding a false positive. In the context of the heuristic, this helps to quantify the risk of incorrectly attributing a size bonus when no further inlining takes place. The higher the precision, the less likely it is to inline a callee because it received an unjustified size bonus.

$$p = \frac{t_p}{t_p + f_p} \tag{4.6}$$

Recall measures how often further inlining was predicted correctly. This quantifies the risk of missing opportunities for further inlining. The higher the recall, the more likely it is to exploit all opportunities for further inlining. For the proposed heuristic, low recall is caused by incomplete dynamic call-graph information. If the available call-graph information was complete, recall for the proposed heuristic would be perfect at a value of 1.

$$r = \frac{t_p}{t_p + f_n} \tag{4.7}$$

In addition to the accuracy, precision and recall statistics, the rate with which further inlining on

objects of a similar type to that of the precise and extant arguments actually occurs can be computed. The set of Equations 4.8 shows how this rate is computed.

$$\begin{aligned} \mu_{\text{extant}} &= \frac{|\{e \in E' \mid \mathscr{P}_{\text{extant}}(e)\}|}{|E'|} \\ \mu_{\text{precise}} &= \frac{|\{p \in P' \mid \mathscr{P}_{\text{precise}}(p)\}|}{|P'|} \end{aligned} \tag{4.8}$$

4.3.1 Accuracy of the Unmodified Heuristic

The unmodified heuristic used by the optimizing compiler always attributes a size-estimate bonus for extant and precise arguments. Using the results of the accuracy metric described above, the accuracy of the unmodified heuristic can be computed as well. This is achieved by recomputing the binary classification under the assumption that further inlining on precise or extant arguments is always predicted, and a negative prediction is never made. This results in the Set of Equations 4.9.

$$\begin{aligned} t'_p &= t_p + f_n \\ f'_p &= t_n + f_p \\ t'_n &= 0 \\ f'_n &= 0 \end{aligned} \tag{4.9}$$

Using this definition, accuracy, precision and recall can be computed for the unmodified heuristic as shown in the set of Equations 4.10.

$$\begin{aligned} a &= \frac{t'_p + t'_n}{t'_p + f'_p + t'_n + f'_n} \\ &= \frac{t_p + f_n}{t_p + f_n + t_n + f_p} \\ p &= \frac{t'_p}{t'_p + f'_p} \\ &= \frac{t_p + f_n}{t_p + f_n + t_n + f_p} \\ r &= \frac{t'_p}{t'_p + f'_n} \\ &= \frac{t_p + f_n}{t_p + f_n} \\ &= 1 \end{aligned} \tag{4.10}$$

For the unmodified heuristic, recall is perfect at 1 because the heuristic always predicts that further inlining will occur. The metrics accuracy and precision compute to the same value. This makes sense because accuracy quantifies the rate of all mispredictions while precision quantifies the rate of mispredictions that attribute the bonus if none should be given. The case that no further inlining is predicted but further inlining does occur never happens. Because of this, the kind of misprediction measured by precision is the only kind of misprediction made by the unmodified compiler. Therefore, accuracy and precision are necessarily equivalent for the unmodified heuristic.

4.3.2 Accuracy of a Random Heuristic

The unmodified heuristic is non-selective, since it alway predicts further inlining and never predicts the opposite. To assess the quality of the proposed heuristic, it makes sense to compare it to another heuristic that actually makes selective predictions. To achieve this, a hypothetical random heuristic is constructed. Whenever this heuristic encounters a precise or extant argument, it uses a random number generator to decide if a bonus should be granted. The probability of granting the bonus is μ_{precise} or μ_{extant}, respectively. In addition to the probability, the absolute number n_p of arguments for which further inlining occurred, and the absolute number n_n of arguments for which further inlining did not occur is required to compute the expected value for the four classes in Table 4.4. The Set of Equations 4.11 shows how n_p and n_n are computed for extant benefits, the Set of Equations 4.12 shows the same for the precise case.

$$\begin{aligned} n_p &= \left|\{e \in E' \mid \quad \mathscr{P}_{\text{extant}}(e)\}\right| \\ n_n &= \left|\{e \in E' \mid \neg\mathscr{P}_{\text{extant}}(e)\}\right| \end{aligned} \tag{4.11}$$

$$\begin{aligned} n_p &= \left|\{p \in P' \mid \quad \mathscr{P}_{\text{precise}}(e)\}\right| \\ n_n &= \left|\{p \in P' \mid \neg\mathscr{P}_{\text{precise}}(e)\}\right| \end{aligned} \tag{4.12}$$

Recalling the Set of Equations 4.8, the probability μ can be rewritten in terms of n_p and n_n as shown in the set of Equations 4.13.

$$\begin{aligned} \mu &= \frac{n_p}{n_p + n_n} \\ 1 - \mu &= \frac{n_n}{n_p + n_n} \end{aligned} \tag{4.13}$$

Using n_p, n_n, and the probability μ, the expected value for the four classes can be computed as shown in the set of Equations 4.14.

$$\begin{aligned} t_p &= n_p\mu \\ f_p &= n_n\mu \\ t_n &= n_n(1-\mu) \\ f_n &= n_p(1-\mu) \end{aligned} \tag{4.14}$$

Finally, the accuracy statistic can be computed by substituting Equation 4.14 into Equation 4.5, as shown in Equation 4.15.

$$
\begin{aligned}
a &= \frac{t_p + t_n}{t_p + f_p + t_n + f_n} \\
&= \frac{n_p\mu + n_n(1-\mu)}{n_p\mu + n_n\mu + n_n(1-\mu) + n_p(1-\mu)} && \text{by (4.14)} \\
&= \frac{n_p\mu + n_n(1-\mu)}{(n_p + n_n)\mu + (n_p + n_n)(1-\mu)} \\
&= \frac{n_p\mu + n_n(1-\mu)}{(n_p + n_n)(\mu + (1-\mu))} \\
&= \frac{n_p\mu + n_n(1-\mu)}{n_p + n_n} \\
&= \frac{n_p}{n_p + n_n}\mu + \frac{n_n}{n_p + n_n}(1-\mu) \\
&= \mu^2 + (1-\mu)^2 && \text{by (4.13)} \\
&= 2\mu^2 - 2\mu + 1
\end{aligned}
\tag{4.15}
$$

Precision for the random heuristic can be computed as shown in Equation 4.16.

$$
\begin{aligned}
p &= \frac{t_p}{t_p + f_p} \\
&= \frac{n_p\mu}{n_p\mu + n_n\mu} && \text{by (4.14)} \\
&= \frac{n_p\mu}{(n_p + n_n)\mu} \\
&= \mu && \text{by (4.13)}
\end{aligned}
\tag{4.16}
$$

Equation 4.17 shows the computation for the recall statistic of the random heuristic.

$$
\begin{aligned}
r &= \frac{t_p}{t_p + f_n} \\
&= \frac{n_p\mu}{n_p\mu + n_p(1-\mu)} && \text{by (4.14)} \\
&= \frac{n_p\mu}{n_p(\mu + 1 - \mu)} \\
&= \mu
\end{aligned}
\tag{4.17}
$$

Surprisingly, precision and recall both compute to μ for the random heuristic. Precision measures the rate of correct predictions of further inlining in all predictions of further inlining. The probability with which the random heuristic predicts further inlining for an argument is μ, regardless if it is part of the n_p or the n_n class. This means that the percentage of correct predictions depends only on the amount of arguments in the n_p class compared to all arguments, which is described by the ratio μ. Recall measures how many arguments from the n_p class are correctly identified as such. When the random heuristic encounters such an argument, it will identify the argument as an argument for which further inlining will occur with a probability of μ. Because of this, recall is also equivalent to μ.

4.3.3 Setup

The selected benchmarks were executed using replay compilation with ten different compilation plans per benchmark. The predictions of the proposed heuristic and all inlining decisions made by the optimizing compiler were recorded. All inlining decisions in which a method of the virtual machine is the callee or the caller, and all inlining decisions in which the extant or precise argument is a virtual machine class were discarded. Since a production configuration of Jikes RVM was used, almost all VM code is precompiled. The methods that are still compiled at runtime are concerned mostly with providing runtime services such as object allocation. Since inlining of such methods is controlled using annotations, predictions that influence the cost-benefit tradeoff for such methods are irrelevant. Therefore, determining the prediction accuracy for these methods does not make sense. This means that the accuracy statistics include only benchmark methods and JRE methods.

Based on the recorded data, the sizes of the true positive, false positive, true negative and false negative classes were computed for all invocations of the benchmarks.

The compiler was configured to ignore the predictions of the heuristic when making inlining decisions, in essence deciding inlining in the same way the unmodified heuristic does. This ensures that the observed behavior that the proposed heuristic is compared against is not influenced by the proposed heuristic itself.

The subset of inlining decisions that are evaluated in this setup influences the outcome of the experiment. One possible approach would have been to force inlining of all call sites. This would make it possible to evaluate the predictions of the heuristic for a very large set of inlining problems. However, this would mean including inlining decisions about otherwise undesirable inlining targets, such as methods called from infrequently executed code. This in turn would result in a skewed representation of the working accuracy of the heuristic. By using the default configuration of the optimizing compiler, it is ensured that the set of evaluated inlining problems closely resembles those inlining problems encountered in a production run of the virtual machine.

		Accuracy (%)			Precision (%)			Recall (%)		
Benchmark	μ (%)	Proposed	Random	Unmodified	Proposed	Random	Unmodified	Proposed	Random	Unmodified
antlr	7.92	94.19	85.20	7.92	88.68	7.92	7.92	27.72↓	7.92	100.00
bloat	12.81	93.31	77.61	12.81	88.01	12.81	12.81	52.11	12.81	100.00
chart	14.30	91.70	75.47	14.30	84.22	14.30	14.30	47.40	14.30	100.00
fop	17.54	94.99	70.90	17.54	90.23	17.54	17.54	80.63	17.54	100.00
hsqldb	11.97	94.06	78.81	11.97	91.11	11.97	11.97	53.44	11.97	100.00
jython	19.93↑	89.94↓	68.00↓	19.93↑	85.21	19.93↑	19.93↑	55.79	19.93↑	100.00
luindex	10.38	93.45	81.36	10.38	91.85↑	10.38	10.38	36.20	10.38	100.00
lusearch	7.25↓	94.46	86.34↑	7.25↓	65.13↓	7.25↓	7.25↓	52.16	7.25↓	100.00
pmd	14.80	97.47↑	74.76	14.80	90.57	14.80	14.80	92.58↑	14.80	100.00
xalan	13.34	91.51	76.82	13.34	81.53	13.34	13.34	41.02	13.34	100.00

↓ = lowest,↑ = highest

Table 4.5: Accuracy of the proposed heuristic, the random heuristic, and the unmodified heuristic

4.3.4 Results

Table 4.5 shows the statistics computed in the progress of the accuracy evaluation. The first numeric column shows the overall μ value per benchmark. Looking at the row for lusearch shows that in this benchmark, further inlining occurred on only 7.25 % of extant and precise arguments (or non-argument objects of similar type). On the opposite end, the further inlining rate for extant and precise arguments is highest for jython at 19.93 %. Note that the values of the μ column are repeated in the accuracy and precision columns of the unmodified heuristic and in the precision and recall columns of the random heuristic. For the random heuristic, it was shown in Section 4.3.2 that precision and recall equal μ. For the unmodified heuristic, some elaboration is needed. The unmodified heuristic always predicts further inlining for extant or precise arguments. The μ percentage states the amount of extant or precise arguments for which inlining actually occurs. The unmodified heuristic is correct in exactly this percentage of cases. Because of this, accuracy is equivalent to μ for the unmodified heuristic.

In Section 4.3.1, it was argued that accuracy and precision are the same for the unmodified heuristic. The μ statistic shows that in most cases, the sole presence of extant or precise arguments does not mean that further inlining on these arguments occurs. This means that the size bonuses assigned by the unmodified heuristic with the rationale that precise or extant arguments enable guardless inlining are unjustified in at least 80 % of all cases.

The accuracy columns of Table 4.5 show an accuracy range of 89.94 % for jython to 97.47 % for pmd. The accuracy of the random heuristic ranges from 68 % for jython to 86.34 % for lusearch. The accuracy of the random heuristic is always lower than the accuracy of the proposed heuristic. The accuracy of the unmodified heuristic is very low at a maximum of 19.93 %. This shows that the proposed heuristic has the best overall prediction quality.

The precision of the proposed heuristic ranges from 65.13 % for lusearch to 91.84 % for luindex. For all benchmarks except lusearch, precision is better than 80 %. At the same time, precision is in the range of the μ percentage of less than 20 % for the random heuristic and the unmodified heuristic. This shows that the proposed heuristic eliminates most arguments for which further inlining does not occur.

The recall statistic of the proposed heuristic is not as good as its precision. Recall ranges from 27.72 % for antlr to 92.58 % for pmd. For the proposed heuristic, low recall is caused by insufficient call-graph completeness. This may be an explanation for the low recall value for antlr. Since call-graph edges are sampled at fixed time intervals and antlr has the lowest total running time of all benchmarks, the number of total call-graph samples for antlr is also lowest for all benchmarks. It is likely that this leads to poor call-graph quality which in turn leads to poor recall performance of the proposed heuristic for antlr. For the proposed heuristic, recall performance is not as good as accuracy and precision. Low recall means that a significant amount of opportunities for extant and precise benefits are probably missed. This is problematic because not reaping such benefits is worse than inlining some methods for which further inlining does not occur. However, recall is still much better than for the random heuristic. The proposed heuristic cannot beat the unmodified heuristic in terms of recall. This is to be expected because the unmodified heuristic achieves its perfect recall at the cost of low overall accuracy.

Figure 4.1 shows a visual comparison of the accuracy statistics for the proposed heuristic, the unmodified heuristic, and the random heuristic. The error bars show the standard deviation over the

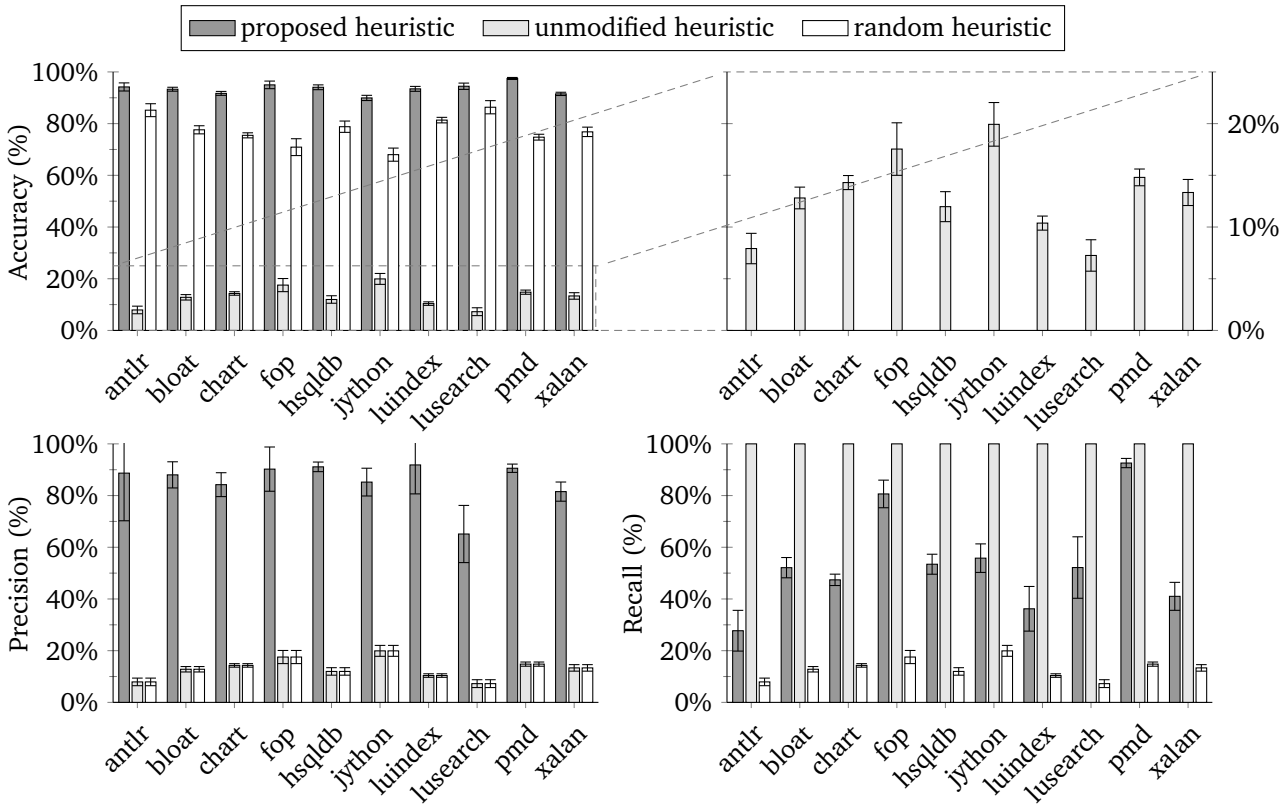

Figure 4.1: Accuracy of the proposed heuristic, the unmodified heuristic, and the random heuristic compared

ten compilation plans for which accuracy was computed. The plots for accuracy and precision show a strong advantage of the proposed heuristic over the baseline heuristic. Accuracy for the random heuristic is much closer to that of the proposed heuristic. In precision terms, the proposed heuristic is clearly better than the unmodified heuristic and the random heuristic. The plot for recall shows that while the proposed heuristic is not nearly as good as the unmodified heuristic, the disadvantage of the proposed heuristic in terms of recall is not as severe as that of the unmodified heuristic in terms of accuracy and precision. Compared to the random heuristic, recall is much better for the proposed heuristic.

4.4 Performance of the Zero Heuristic

The proposed heuristic works by removing the size-estimate bonus attributed for each extant or precise argument when there is no call-graph edge that suggests that a method is invoked on that argument. This is a selective heuristic that attributes the bonus only in some cases. The unmodified heuristic of Jikes RVM always attributes these bonuses. A *zero heuristic* is constructed that represents the other extreme. This heuristic never attributes size bonuses for precise or extant arguments.

Comparing the zero heuristic and the unmodified heuristic provides an understanding of the performance limits in which the proposed heuristic operates. Since the proposed heuristic attributes the size bonuses in some cases, but not in all cases, this should lead to more inlining than using the zero heuristic but less inlining than using the unmodified heuristic.

If it is assumed that more inlining always means more time spent compiling and better code quality, the compile time of the proposed heuristic should be higher than that of the zero heuristic and lower than that of the unmodified heuristic. Similarly, code quality should be best for the unmodified heuristic and worst for the zero heuristic. Optimally, the call-graph-based heuristic should lead to a large saving of compile time while maintaining code quality similar to that of the unmodified heuristic.

Performance was measured for the unmodified heuristic and the zero heuristic using the compiler-replay technique described in Section 4.1.2. For each benchmark, ten compilation plans were measured for each of the heuristics.

4.4.1 Results

Figure 4.3 shows the results of the measurement. The per-benchmark scatter plots show the pure pro-

gram execution time on the y axis and the aggregate compile time of the optimizing compiler and the baseline compiler on the x axis. Both quantities are shown in milliseconds and the unit length is the same for both axes. Each mark represents a single invocation of the benchmark using one of ten compilation plans. The total running time of the benchmark is the sum of both quantities and is visualized by diagonal lines in the plots. Two marks that rest on the same diagonal line have the exact same total running time. The diagonal distance of two lines represents 2.5% of the average running time of the benchmark when executed by the unmodified version of the virtual machine. The corridors between the diagonal lines are a visual help for estimating the difference in total running time for the individual marks. The density of the lines helps to estimate how large the difference between the plans is. The denser the lines are, the more difference there is between the individual measurement results of the different plans and revisions. Marks that are close to the lower left corner of the plot exhibit the best overall performance.

In the first benchmark **antlr**, there are two outliers: one is a measurement result of the unmodified heuristic, the other represents the zero heuristic. For both results, compile time is about twice as long as for the other plans. Program time is significantly reduced, resulting in a total running time that is similar to that of the other runs. Both outliers are the result of running the benchmark with compilation plan 1. The fact that the first plan for each heuristic is affected suggests caching issues as a possible explanation. This is contradicted by the fact that code quality is better for both plans. This suggests that the outliers are caused by the structure of compilation plan 1.

The plot to the right of the first plot shows the same results, zoomed in on the cluster formed by compilation plans 2 to 10. The results for the zero heuristic and the unmodified heuristic are not clearly separated. Compile time is similar for both heuristics. However, it can be seen that pure program time is worse for the zero heuristic. This suggests that overall performance is better for the unmodified heuristic.

The next plot shows the measurement results for **bloat**. Here, the results are more conclusive. There are two clusters of measurement results that differ in compile time. Six of the results for the unmodified heuristic have a compile time that is clearly greater than that of the zero heuristic. The other four results lie within a cluster formed together with the results of the zero heuristic. No difference in program time is apparent from the plots. The zero heuristic results in large savings of compile time for most of the plans, while code quality stays the same. This results in a clearly visible performance improvement of the zero heuristic for this benchmark.

The scatter plot for **chart** again shows an outlier for the unmodified heuristic. Inspection of the raw data reveals that the outlier mark represents compilation plan 1, which is the first instance of chart that is run while benchmarking. The difference for this plan is mostly in execution time, not in compile time. Chart has significant external dependencies. On Linux, it depends on the *X11 Window System* and the *GTK+* toolkit library, among others. While the measurement script loads the window system before executing any benchmarks, the native shared libraries are loaded when they are first used by the benchmarks. This suggests that loading the external native libraries causes the increase in total running time. This hypothesis was verified by benchmarking a different plan first, which resulted in a similar increase in compile time. When the measurement was repeated directly after that, the effect was not observed. This leads to the conclusion that loading the native libraries caused the outlier. Repeating all measurements for chart was not feasible due to time constraints. Because of this, all results for compilation plan 1 were discarded when computing mean performance metrics for chart.

The plot to the right is a zoomed-in version of the original plot for chart. Here, differences in program time result in clear clustering. Program time is approximately 5% worse for the zero heuristic. Compile time is slightly better for the zero heuristic, but the difference is hardly discernible. Since chart spends only about 12% of its time compiling code (cf. Table 4.1), savings in compile time only make a small difference, while reduced code quality results in a big loss of overall performance. Because of this, performance is clearly worse for the zero heuristic.

The next plot shows the results for **fop**. The variance in compile time and program time between measurements of the same heuristic is large. The result is inconclusive as to the difference of the two versions of the heuristic.

The plot for **hsqldb** shows a horizontal separation of the zero heuristic and the unmodified heuristic. Most results of the unmodified heuristic show higher compile time. Program time is clearly worse for three of the zero heuristic-results. For the other results, no clear vertical clustering is apparent. Looking at the number of marks in each of the diagonal corridors suggests that the compile time savings achieved with the zero heuristic simply compensate for the additional program time caused by lower code quality. There is no clear difference in overall performance.

The results for **jython** show that program time varies for the individual plans, while the variance in compile time is smaller. Program time is slightly

worse for most measurement results of the zero heuristic, while no difference in compile time is apparent between the two heuristics. Because of this, total performance is better for the unmodified heuristic.

The plot for **luindex** shows that compile time varies strongly between the different plans for both of the heuristics. Program time is good for most of the measurement results of the unmodified heuristic. For the zero heuristic, program time is worse and the difference in program time between the individual plans is higher than for the unmodified heuristic. Overall, performance is worse for the zero heuristic.

The result for **lusearch** is similar to that of luindex. The difference here is that there is also considerable variance in program time for the unmodified heuristic. The program time of the zero heuristic is still worse than for the unmodified heuristic. For most of the plans, compile time is better for the zero heuristic. Overall, performance is better for the unmodified heuristic. However, the difference is not as large as for luindex.

The scatter plot for **pmd** shows a clear separation on the horizontal axis. Compile time is clearly better for the zero heuristic. There is considerable inter-plan variance of program time. Still, program time is slightly better for the unmodified heuristic. Overall, the savings in compile time overcompensate the difference in program time. As a result, the zero heuristic performs better than the unmodified heuristic.

The plot for the last benchmark **xalan** shows a large variance in program time that is independent of the used heuristic. Variance for compile time is much lower. The plot shows that most of the marks for the zero heuristic are placed to the left of the marks for the unmodified heuristic. However, this small difference in compile time is dwarfed by the large differences in program time across all plans. As a result, the plot shows no discernible difference in total performance for both heuristics.

Figure 4.2 shows the mean total-running-time speedup caused by the zero heuristic for all benchmarks. The per-benchmark speedup is the geometric mean of the per-plan speedup of the zero heuristic. In addition to computing the speedup, a paired t-test of the difference in total running time was performed for all plans. Benchmarks for which the t-test yields a significance level $\alpha < 0.05$ are shown as gray bars in the diagram. The only benchmarks for which there is a significant difference in total running time are bloat, chart and luindex. The zero heuristic is 8.01% faster for bloat, 6.36% slower for chart and 1.73% slower for luindex. For all other benchmarks, the probability that the observed difference in total running time is caused by effects other than the difference of the two heuristics is higher than 5%.

Overall, the experiment shows that attributing bonuses for precise and extant arguments has different effects for the benchmarks. The benchmark bloat, which spends about 21% of its time compiling (cf. Table 4.1), profits greatly from the reduction in compile time caused by not granting the bonuses, while code quality is preserved. The benchmark pmd shows a similar, but weaker effect that is not statistically significant. On the other end of the spectrum, the benchmark chart, which spends only about 12% of its time compiling, shows a large reduction in code quality while there is no worthwhile saving of compile time. For the benchmark luindex, the reduction in code quality is not as pronounced as for chart, however the aggregate effect of the zero heuristic is again worse performance. The benchmarks hsqldb and lusearch show some savings in compile time that are compensated by worse code quality. For the benchmarks antlr, fop, jython and xalan, the differences between the heuristics are dwarfed by other performance effects.

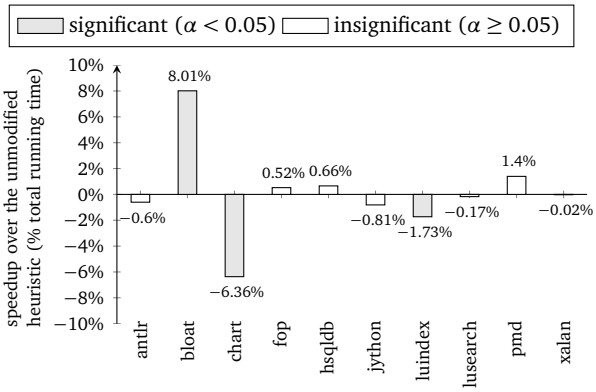

Figure 4.2: The total-running-time speedup of the zero heuristic over the unmodified heuristic

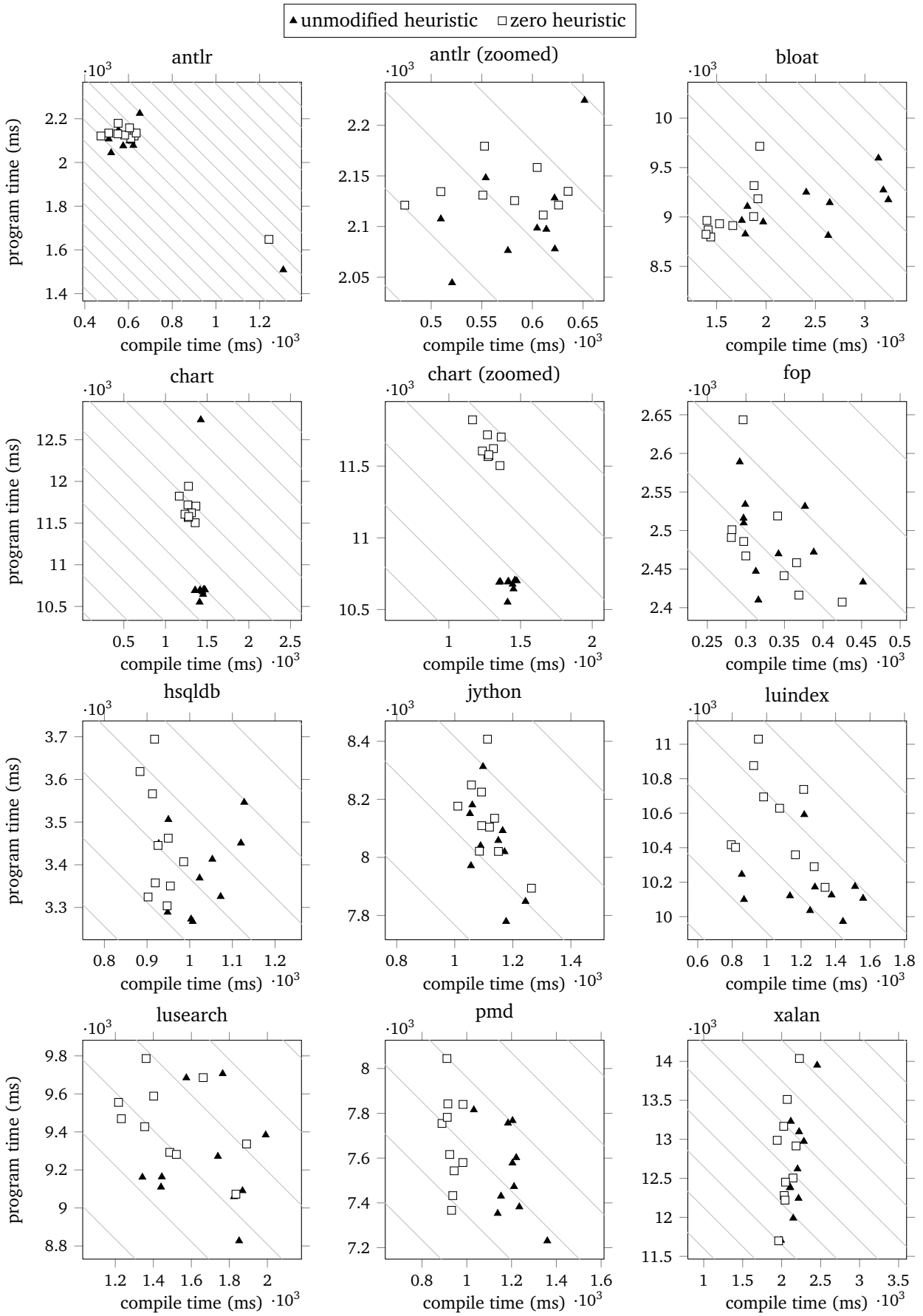

Figure 4.3: The effect of the zero heuristic on compile time and pure program time. Each mark represents the result of running the benchmark with a single compilation plan (out of ten). The diagonal lines visualize total running time. Two marks on the same line have the same total running time. The distance between two lines represents 2.5% of the average total running time for the unmodified heuristic.

4.5 Performance of the Proposed Heuristic

Benchmarking the zero heuristic provided a first impression of the performance impact of assigning bonuses for precise and extant arguments. In the next step, the proposed heuristic is evaluated in the same way. The goal of the proposed heuristic is to reduce compile time as much as possible while retaining as much code quality as possible.

4.5.1 Results

Figure 4.5 shows the same kind of scatter plot that was used to evaluate the zero heuristic in Section 4.4. The marks show the measurement results for the proposed heuristic and the unmodified heuristic. Again, diagonal lines spaced at a distance of 2.5% of the average total running time of the unmodified heuristic are used as a visual aid.

The plot for **antlr** again shows two outliers for compilation plan 1. Surprisingly, the mark for the proposed heuristic is to the lower right of the mark for the unmodified heuristic. This means that the proposed heuristic caused longer compile time and better code quality for this plan. Since the proposed heuristic should in theory inline less methods, worse code quality and shorter compile time is the expected result. It is unclear what causes this discrepancy.

The zoomed-in version of the plot for antlr shows a result that is similar to that of the zero heuristic. Program time is slightly worse for the proposed heuristic while compile time is unchanged. The inter-plan differences are larger than the differences of the two heuristics.

In the plot for **bloat**, there is again a cluster of results for the unmodified heuristic at the right side of the plot. Using the proposed heuristic, compile time is significantly reduced while there is no evident difference in code quality between the two heuristics.

The result for the benchmark **chart** is very different from the observations regarding the zero heuristic. Ignoring the outlier of the unmodified heuristic, the magnified plot shows that there is now only a small difference in program time. There are three plans for which pure execution time is clearly longer for the proposed heuristic. For the other plans, code quality is on par with that of the unmodified heuristic. As a result, the reduction in compile time for the proposed heuristic becomes visible in the plot.

The plot for **fop** shows a more compact distribution of compile time and program time for the proposed heuristic than for the unmodified heuristic. Compile time is slightly better for the proposed heuristic, while code quality is worse. This differs from the results of the zero heuristic, where there was no immediately visible difference between the heuristics.

The results for **hsqldb** show a visible reduction in compile time for the proposed heuristic. Program time is worse for at least three of the compilation plans. The overall performance of hsqldb is largely unaffected by the proposed heuristic. Figure 4.4 confirms that there is no significant difference in total running time for hsqldb.

The benchmark **jython** shows no visually apparent difference in compile time. At the same time, there is a large inter-plan variance in program time. On average, performance is slightly worse for the proposed heuristic. This result is very similar to the result for the zero heuristic.

The plot for **luindex** is largely inconclusive. The two marks with the worst compile time are results of the unmodified heuristic. The mark with the worst program time is a result of the proposed heuristic. Overall, there is only a small insignificant difference in running time.

The plot for **lusearch** shows that the proposed heuristic reduces compile time, while code quality is only worse in some cases. Overall, the number of points representing the unmodified heuristic in the lowest corridor suggests that there is an improvement in performance.

For the benchmark **pmd**, the proposed heuristic results in a reduction of compile time. At the same time, program time increases. There is no visible difference in overall performance for the two heuristics.

The plot for **xalan** again shows large differences in program time between the compilation plans, with no visible clustering that suggests a difference in program time for the compared heuristics. A reduction in compile time is visible, but the effect is dwarfed by program time differences.

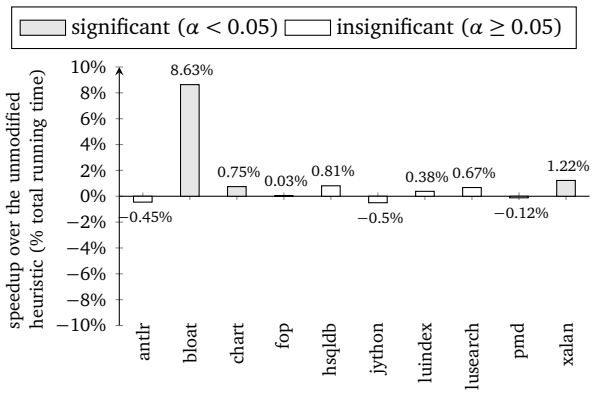

Figure 4.4: The total-running-time speedup of the proposed heuristic over the unmodified heuristic

Figure 4.4 shows the speedup achieved by employing the proposed heuristic. There is a significant difference in total running time for the benchmarks bloat, chart and xalan. The benchmark bloat

is 8.63% faster, which is similar to the speedup achieved by the zero oracle. While chart is 6.36% slower when executed using the zero oracle, using the proposed heuristic leads to a small significant speedup of 0.75%. For chart, the comparison of the zero heuristic and the proposed heuristic shows that smart assignment of bonuses in the proposed heuristic succeeds in reducing compile time while preserving code quality. For the benchmark xalan, there is no difference in total running time between the zero heuristic and the unmodified heuristic. In contrast, using the proposed heuristic results in a speedup of 1.22%.

For the benchmarks antlr, fop, hsqldb, jython, luindex and lusearch, there is no significant difference in running time when comparing the proposed heuristic to the unmodified heuristic. For the zero heuristic, luindex performance is degraded by 1.73%. For the proposed heuristic, no such performance degradation is observed. The proposed heuristic only shows a small, statistically insignificant performance degradation for antlr, jython and pmd.

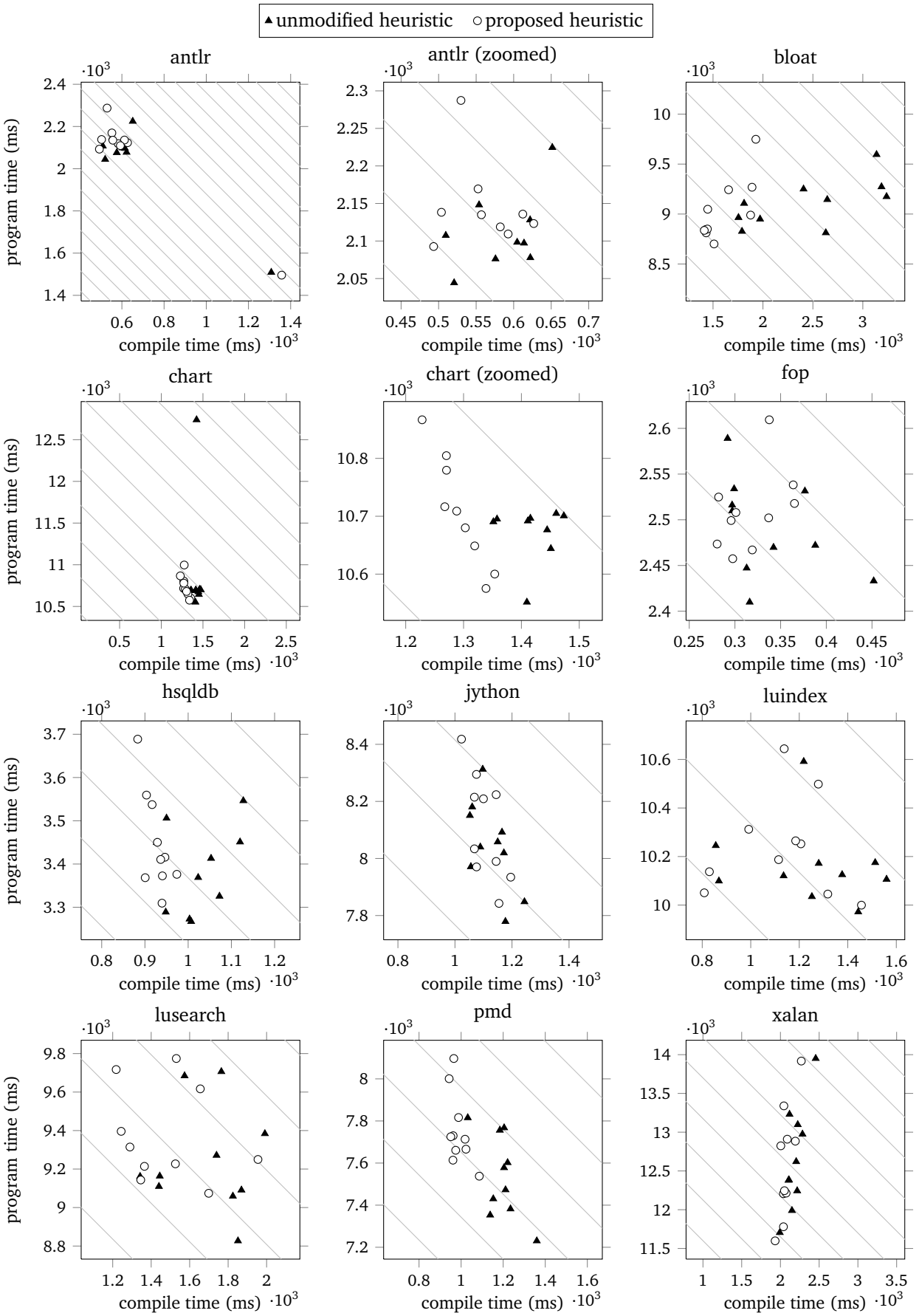

Figure 4.5: The effect of the proposed heuristic on compile time and pure program time. Each mark represents the result of running the benchmark with a single compilation plan (out of ten). The diagonal lines visualize total running time. Two marks on the same line have the same total running time. The distance between two lines represents 2.5% of the average total running time for the unmodified heuristic.

4.6 Performance of the Deep-Inlining Heuristic

In the previous sections, the proposed call-graph-based heuristic was evaluated by comparing its performance to that of the unmodified heuristic and the zero heuristic. In Section 3.4.1, a variant of the proposed heuristic was described that traverses the call graph recursively to detect edges that suggest a method being invoked on a precise or extant argument. In this section, this *deep-inlining* variant of the proposed heuristic is evaluated by comparing its performance to that of the non-recursive version of the proposed heuristic and to that of the unmodified heuristic.

4.6.1 Results

Figure 4.7 shows the results of the measurement. For the benchmark **antlr**, there is again an outlier for each of the heuristics. The mark for the deep-inlining heuristic shows increased program time and slightly reduced compile time.

The plot to the right shows a magnification of the area in which the non-outlier marks are located. Comparing the marks for the proposed heuristic and the deep-inlining variant shows that the latter results in an increased program time for most of the compilation plans. Compile time for the deep-inlining variant is similar to that of the proposed heuristic. There is no large difference in total execution time visible.

The plot for **bloat** shows that for a single compilation plan, the deep-inlining variant of the proposed heuristic results in a large reduction of program time. At the same time, compile time is very low for this plan. For most other marks, program time and compile time are similar to that of the non-recursive heuristic. Overall, there is no significant improvement in total running time.

The benchmark **chart** again shows an outlier for the unmodified heuristic. The corresponding marks for the two versions of the proposed heuristic also show increased program time. However, the effect is much weaker for these heuristics.

The plot to the right is zoomed in on the non-outlier results for chart. The effect of the deep-inlining heuristic and the proposed heuristic is very similar in terms of program time and compile time.

For the benchmark **fop**, the deep-inlining variant of the proposed heuristic shows worse program time when compared to the non-recursive heuristic. Compile time is similar for both non-recursive heuristics. Overall running time is worst for the deep-inlining variant of the proposed heuristic

The plot for **hsqldb** is inconclusive. Compile time and program time are similar for the two proposed heuristics. For **jython**, **pmd** and **xalan**, there is no visually discernible difference for the heuristics either.

For the benchmark **luindex**, the plot shows that there is an increase in program time for the deep-inlining variant of the heuristic, while compile time is similar. For **lusearch**, program time also increases with no visible difference in compile time.

For most benchmarks, the deep-inlining variant of the heuristic behaves very similarly to the non-recursive proposed heuristic. The deep-inlining heuristic assigns the bonus in all cases in which the non-recursive heuristic does. In addition to that, it also assigns the bonus when an extant-induced or precise-induced edge (cf. Definition 3.6 and Definition 3.5) is discovered by recursively traversing the call graph. This means that this heuristic inlines all call sites inlined by the non-recursive heuristic and some additional ones that are not inlined by the non-recursive heuristic. Under the assumption that inlining improves code quality at the cost of increasing compile time, the deep-inlining heuristic should yield better code quality and worse compile time when compared to the non-recursive heuristic. Surprisingly, this is not the case for most benchmarks. The benchmarks antlr, fop, and luindex show this very clearly. In those benchmarks, the deep-inlining heuristic results in worse code quality, while there is no pronounced change in compile time. On the other hand, the benchmark bloat does show improved program time for some of the plans. However, there is no visible change in compile time for bloat.

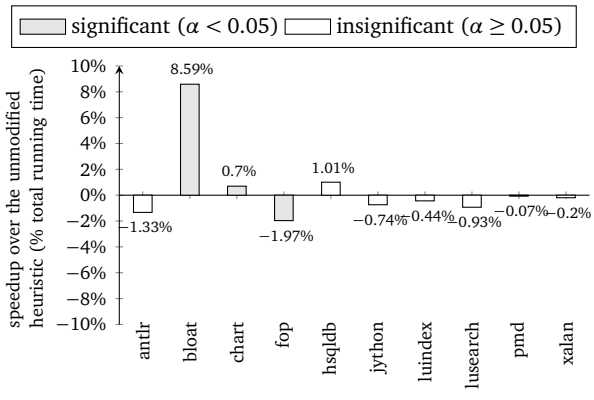

Figure 4.6: The total-running-time speedup of the deep-inlining variant of the proposed heuristic over the unmodified heuristic

Figure 4.6 shows the speedup of the deep-inlining variant of the proposed heuristic compared to the unmodified heuristic. There is a significant difference in total running time for the benchmarks bloat, chart and fop. The deep-inlining heuristic achieves a mean speedup of 8.59% for bloat. This is essentially the same as the 8.63% speedup caused by the non-recursive heuristic. For chart, the speedup of 0.7% is again equivalent to that of the

non-recursive heuristic. For fop, the deep-inlining heuristic is significantly worse at a slowdown of 1.97%. While the non-recursive heuristic shows a significant 1.22% speedup for xalan, there deep-inlining heuristic shows an insignificant slowdown of 0.2%. In total, all benchmarks except bloat, chart and hsqldb show a negative speedup for the deep-inlining heuristic.

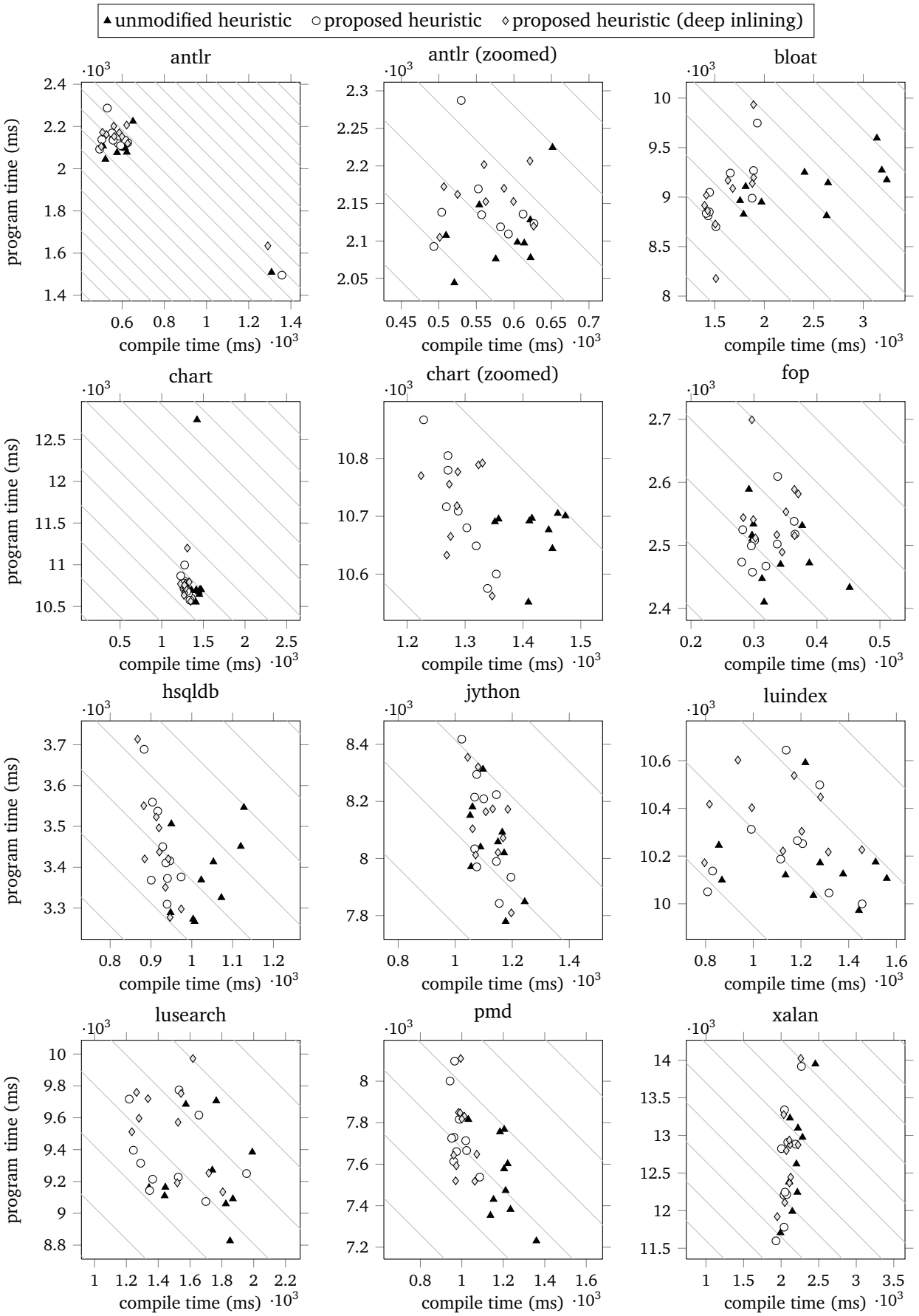

Figure 4.7: The effect of the deep-inlining variant of the proposed heuristic on compile time and pure program time. Each mark represents the result of running the benchmark with a single compilation plan (out of ten). The diagonal lines visualize total running time. Two marks on the same line have the same total running time. The distance between two lines represents 2.5% of the average total running time for the unmodified heuristic.

4.7 Performance of the Conductive-Call-Graph Heuristic

Figure 4.9 shows the results of the performance measurement for the conductive-call-graph variant of the proposed heuristic that was described in Section 3.4.2.

Looking at the first plot for **antlr**, there is again an outlier for compilation plan 1 of all heuristics. The zoomed-in plot shows that program time is slightly worse for the conductive-call-graph variant of the proposed heuristic, while compile time is essentially the same as for the proposed heuristic.

The results for **bloat** show that program time is better for the conductive-call-graph variant than for the proposed heuristic. This gain is achieved at the cost of increasing compile time. However, the compile time of the variant heuristic is still better than that of the unmodified heuristic. Over all three heuristics, program time is best for the variant heuristic.

The plot for **chart** shows that the conductive-call-graph heuristic results in a program time that is slightly better than that of the proposed heuristic but still worse than that of the unmodified heuristic. Compile time is similar to that of the originally-proposed heuristic.

For **fop**, the conductive-call-graph heuristic actually results in an increased program time and compile time when compared to the proposed heuristic. Here, the additional inlining caused by the variant heuristic seems to have an adverse effect on code quality. Overall, fop is the worst case for the variant heuristic.

The plot for **hsqldb** shows no clear difference in compile time and program time for the variant heuristic and the proposed heuristic. Compile time is better than that of the unmodified heuristic for both.

The effect of the conductive-call-graph heuristic on **jython** is an increase in compile time and a reduction in program time. In contrast to bloat, compile time for the conductive-call-graph heuristic is higher than that for the unmodified heuristic. This means that the conductive-call-graph heuristic leads to noticeably more inlining than the unmodified heuristic in this case.

The plot for **luindex** shows that program time for the conductive-call-graph heuristic is better than that for the other two heuristics in most cases. At the same time, compile time for the variant heuristic is similar to that of the proposed heuristic and better than that of the unmodified heuristic in most cases.

For **lusearch**, the program time caused by the conductive-call-graph heuristic is similar to that for the other heuristics, while compile time is slightly increased in comparison to the proposed heuristic.

The plot for **pmd** shows that the inter-plan differences in compile time and performance are smaller for the conductive-call-graph heuristic than for the other heuristics. Program time is better than that of the proposed heuristic and worse than that of the unmodified heuristic. Compile time for the variant heuristic is worse than for the proposed heuristic and better than for the unmodified heuristic. This results in a similar overall performance for all heuristics.

For **xalan**, program time of the variant heuristic is similar to that of the proposed heuristic. Compile time is similar for the variant heuristic and the proposed heuristic.

The conductive-call-graph heuristic achieves a noticeable increase in code quality for the benchmarks bloat, jython and luindex when compared to the proposed heuristic. For bloat, this improvement is only possible at the cost of increasing compile time. For luindex, an increase in compile time is hardly noticeable. For antlr and fop, program time actually increases for the conductive-call-graph heuristic.

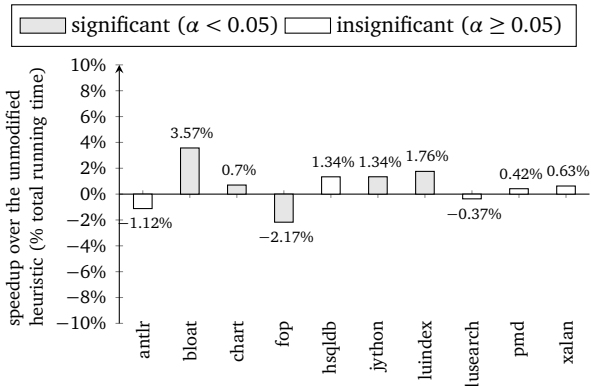

Figure 4.8: The total-running-time speedup of the conductive-call graph-variant of the proposed heuristic over the unmodified heuristic

Figure 4.8 shows the total-running-time speedup for the conductive-call-graph variant of the proposed heuristic when compared to the unmodified heuristic. For bloat, chart, fop, jython and luindex, there is a significant difference in total running time. Performance for bloat is best at a speedup of 3.57 %. However, this speedup is less pronounced than the 8.63 % achieved by the originally-proposed heuristic. This shows that for bloat, the increase in compile time does not justify the achieved improvement in code quality. For chart, the speedup of 0.7 % is equivalent to that achieved by the proposed heuristic. The change for fop is a noticeable degradation at a negative slowdown of 2.17 %. The benchmarks jython (1.34 %) and luindex (1.76 %) perform much better with the conductive-call-graph heuristic than with the proposed heuristic. On the other hand, there is no longer a significant speedup for xalan.

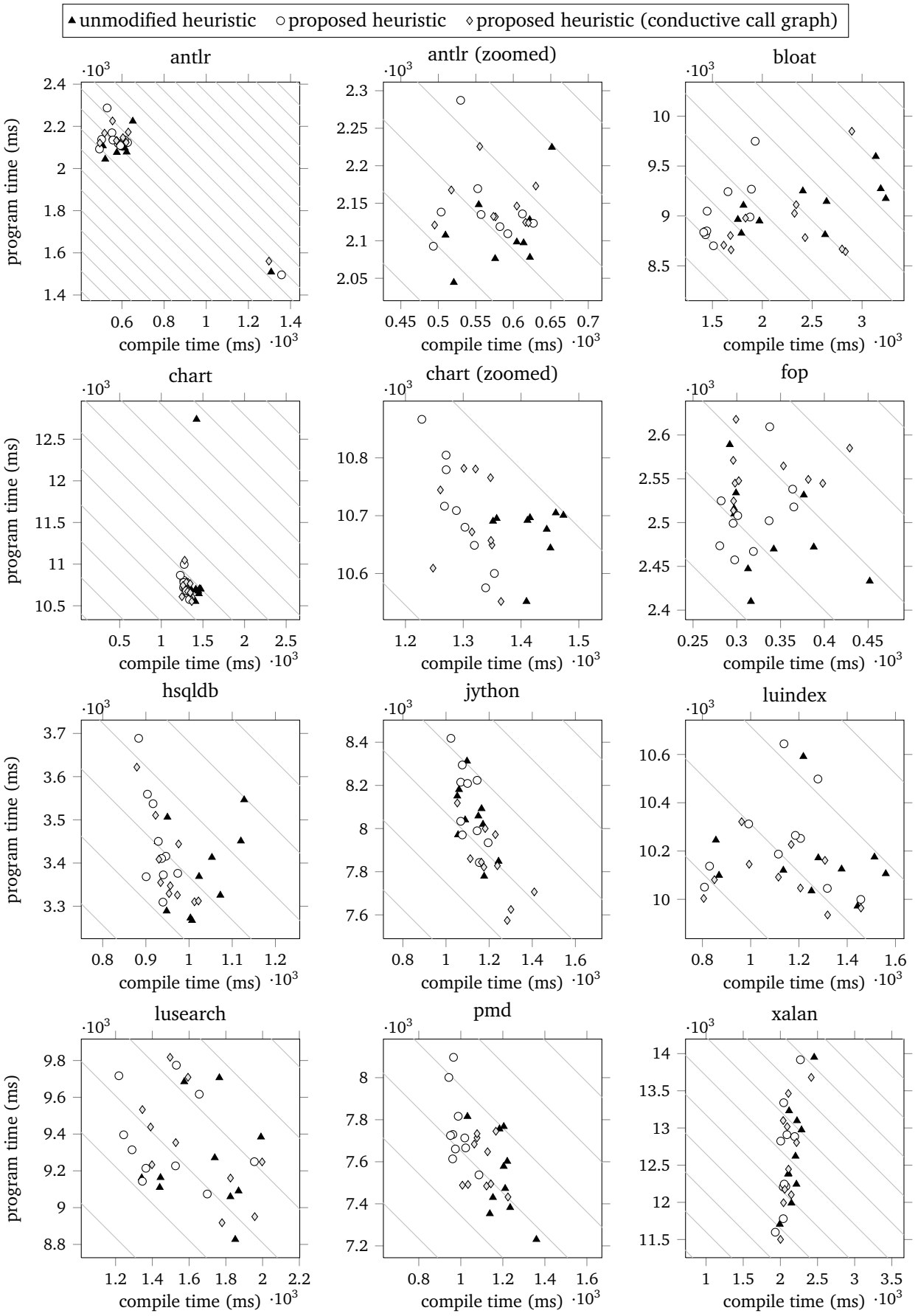

Figure 4.9: The effect of the conductive-call-graph variant of the proposed heuristic on compile time and pure program time. Each mark represents the result of running the benchmark with a single compilation plan (out of ten). The diagonal lines visualize total running time. Two marks on the same line have the same total running time. The distance between two lines represents 2.5% of the average total running time for the unmodified heuristic.

4.8 Performance of the Proposed Heuristic without Replay Compilation

In the previous sections, the performance of the proposed heuristic and the two variant heuristics was evaluated using replay compilation. Overall, the performance of the originally-proposed heuristic is most favorable because it achieves a significant speedup for three out of ten benchmarks and no significant performance degradation for all other benchmarks.

Since replay compilation may bias the results of the performance measurement in one direction or the other (cf. Section 4.1.2), the proposed heuristic is additionally evaluated using a non-replay measurement setup. This also allows using the benchmark **eclipse** that was previously excluded.

Each benchmark was invoked 16 times for the proposed heuristic and the unmodified heuristic. The result of the first invocation was discarded for both heuristics, resulting in a total of 15 measured invocations. Performance was measured in the single-iteration setting, as well as for five and ten iterations (cf. Appendix A). The reported speedup was computed from the average total running time of each of the heuristics. To determine significance, an unpaired t-test was performed on the same input data. This test determines if the averages of the two heuristics are significantly different to a level of $\alpha < 0.05$.

4.8.1 Results

Figure 4.11 shows the results of the non-replay performance measurement. The result for **antlr** shows that compile time is spread out for both heuristics. There is a single outlier for the unmodified heuristic that has significantly higher compile time and code quality at the right end of the plot. In the upper portion of the plot, there is an outlier for the proposed heuristic that has significantly increased program time. Otherwise, there is no clear difference between both heuristics visible.

The plot for **bloat** shows that compile time is much better for the proposed heuristic while program time is worse. There are two invocations that exhibit much higher compile time and strongly improved program time for the unmodified heuristic.

The plot for **chart** shows clear clustering along both axes. The proposed heuristic exhibits increased program time and reduced compile time when compared to the unmodified heuristic. Overall running time is very similar for both heuristics.

The plot for **fop** shows four outliers for the unmodified heuristic that exhibit increased compile time and reduced program time. Otherwise, performance seems to be the same for both heuristics.

The plot for **hsqldb** shows a large range of compile time and program time for both heuristics. Program time is increased for at least two results of the proposed heuristic, while compile time is worst for two results of the unmodified heuristic. Otherwise, there is no clear difference visible when comparing both heuristics.

The plot for **jython** shows an increase in program time and a decrease in compile time for most results of the proposed heuristic. Overall performance seems to be similar for both heuristics.

For **luindex** and **lusearch**, program time is increased and compile time is reduced for the proposed heuristic.

The benchmark **pmd** shows clearly reduced compile time for the proposed heuristic, while program time is equivalent to that of the unmodified heuristic for most invocations. An overall performance gain for the proposed heuristic is visible.

The last benchmark **xalan** shows four outliers with strongly increased program time and strongly reduced compile time. Three of those outliers are representatives of the proposed heuristic. Otherwise, there is no clear difference between the heuristics.

The benchmark **eclipse** that was not measured in the replay setting shows increased program time and reduced compile time for the proposed heuristic.

Even without replay compilation, the proposed heuristic has a visible effect on the relationship between compile time and program time. A reduction in compile time is visible for bloat, chart, fop, jython, luindex, lusearch, pmd, and eclipse. An increase in program time is visible for bloat, chart, fop, hsqldb, jython, luindex, lusearch, pmd, and eclipse.

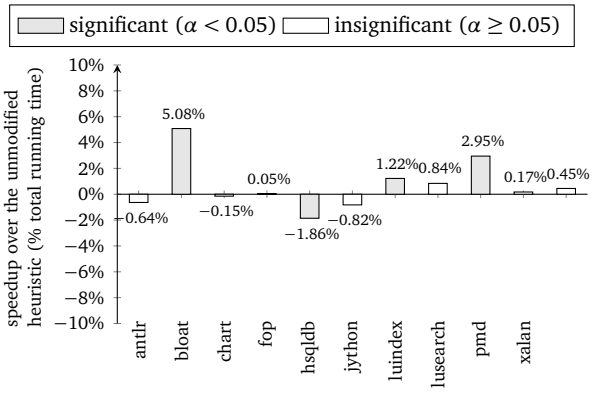

Figure 4.10: The total-running-time speedup of the proposed heuristic over the unmodified heuristic in a setting without replay compilation

Figure 4.10 shows the average speedup in total running time when comparing the proposed heuristic to the unmodified heuristic. Total running time

is significantly different for bloat, hsqldb, luindex and pmd. Of those, hsqldb is the only benchmark that shows a slowdown at -1.86 %. For bloat, the speedup of 5.08 % is smaller than the speedup achieved in the replay setting, but still represents a profitable gain. The speedup of luindex is small at 1.22 %. The benchmark pmd that showed unchanged performance in the replay setting now shows a worthwhile speedup of 2.95 %. For xalan, which ran 1.22 % faster in the replay setting, no change in total running time is observed.

Overall, the proposed heuristic achieves a significant speedup for three benchmarks and shows a significant slowdown only for a single benchmark. It can be concluded that the proposed heuristic has an effect that is noticeable over the noise of other subsystems in the virtual machine. The overall effect of employing the proposed heuristic is still positive when the call graph is sampled live as opposed to being loaded from a pre-recorded profile.

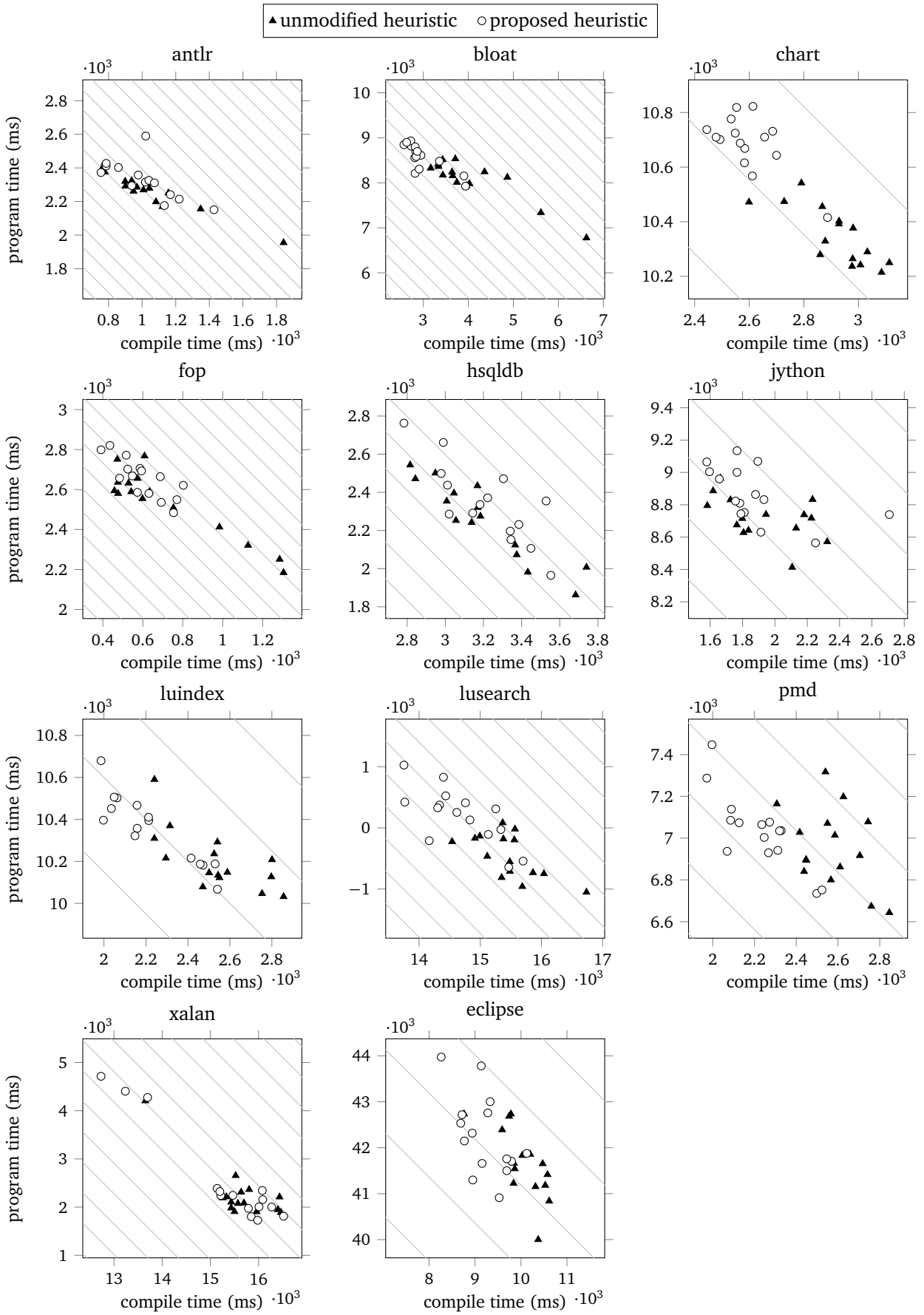

Figure 4.11: The effect of the proposed heuristic on compile time and pure program time in a setting without replay compilation. Each mark represents the result of a single invocation of the benchmark. The diagonal lines visualize total running time. Two marks on the same line have the same total running time. The distance between two lines represents 2.5% of the average total running time for the unmodified heuristic.

4.9 Summary of the Performance Evaluation

Figure 4.12 shows the total running time speedup of all evaluated heuristics when compared to the unmodified heuristic in a setting using replay compilation. In addition to the individual benchmark speedup, two speedup averages are shown for each heuristic. The first average bar shows the arithmetic mean over all significant benchmark speedups. The second average bar shows the arithmetic mean over all benchmark speedups, significant and insignificant.

The results for the zero heuristic in Figure 4.12 show that this heuristic is worst. The significant results average a slowdown of -1.73 %, while the average speedup including insignificant benchmark results is neutral at 0.09 %.

Out of all evaluated heuristics, the original version of the proposed heuristic performs best. The average significant speedup is 3.53 %, the overall average speedup is still positive at 1.14 %.

The deepiinlining variant of the proposed heuristic achieves an average speedup of 2.44 % for the significant results and an overall speedup of 0.46 %. The conductive-call-graph variant of the proposed heuristic achieves a smaller speedup of 1.04 % for the significant results. The overall average speedup is 0.61 % for this heuristic. The diagrams show that of the variant heuristics, the deep-inlining variant performs best on average. On the other hand, the conductive-call-graph variant achieves a significant speedup for 4 benchmarks. The fact that the conductive-call-graph variant is worse on average can be attributed to the substantially smaller speedup achieved for bloat.

Figure 4.13 shows the speedup achieved by the proposed heuristic in a setting without replay compilation. In addition to the single-iteration measurements already shown in Section 4.8, the measurements for long-running loads (cf. Section 4.1.4) simulated by running five, respectively ten iterations of each benchmark are shown. In the single-iteration setting, a significant speedup is achieved for three benchmarks, while a significant slowdown is observed for a single benchmark. The average significant speedup is 1.85 %, the combined average speedup for all benchmarks is 0.66 %. At five iterations, a significant speedup is observed for four benchmarks and no benchmark suffers a significant slowdown. The average significant speedup at five iterations is 3.83 %, the overall average speedup is 1.59 %. At ten iterations, a significant speedup is observed for six benchmarks, and a significant slowdown for a single benchmark. The significant results average a speedup of 3.43 %, overall speedup is at 2.28 %.

The results show that for the simulated long-running loads, the number of benchmarks for which a significant speedup is observed is greater for the five-iteration setting than for the single-iteration setting and even greater for the ten-iteration setting. The five-iteration loads and ten-iteration loads show a more pronounced average speedup than the single-iteration load. It can be concluded that saving compile time due to inlining less methods is still profitable when a single program has been running for a longer period of time. In the observed cases, the positive effect of saving compile time becomes more pronounced for long-running loads.

From the presented performance measurements, it can be concluded that the proposed heuristic has a noticeable effect on most benchmarks. For some benchmarks, savings in compile time are compensated by increased program time. For other benchmarks, the proposed heuristic results in better overall performance, both in a setting that uses replay compilation and when the adaptive optimization system is used. In the latter case, overall performance is worse for some benchmarks. However, average performance increases in all cases.

For the other evaluated heuristics, a positive effect on overall execution time occurs for some benchmarks, while the effect is negative for other benchmarks. It can be concluded that the variant heuristics do not represent improvements over the originally-proposed heuristic in their current form.

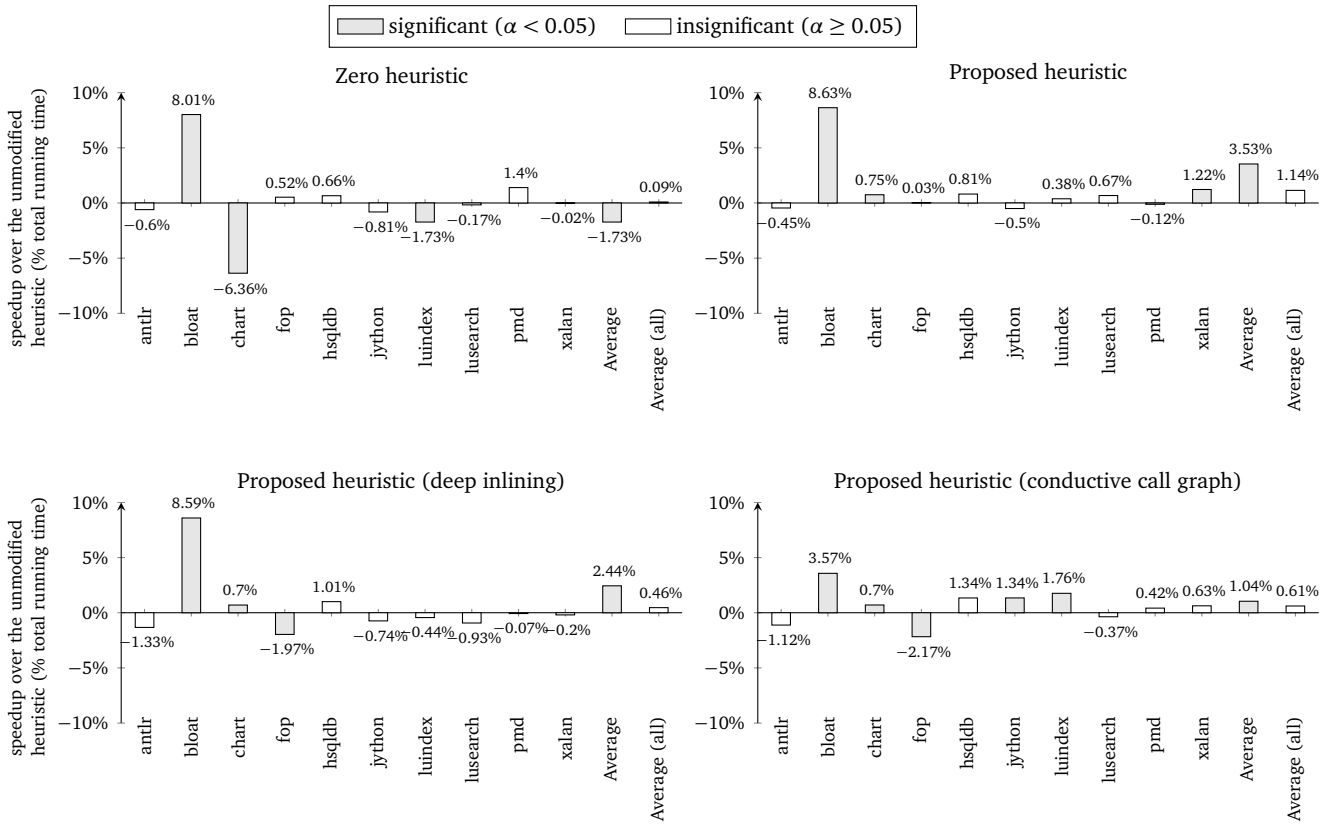

Figure 4.12: The total-running-time speedup of the evaluated heuristics over the unmodified heuristic. The percentages shown represent the mean of the per-plan speedup over ten compilation plans. The gray bars show benchmarks for which there is a statistically significant difference in total running time ($\alpha < 0.05$).

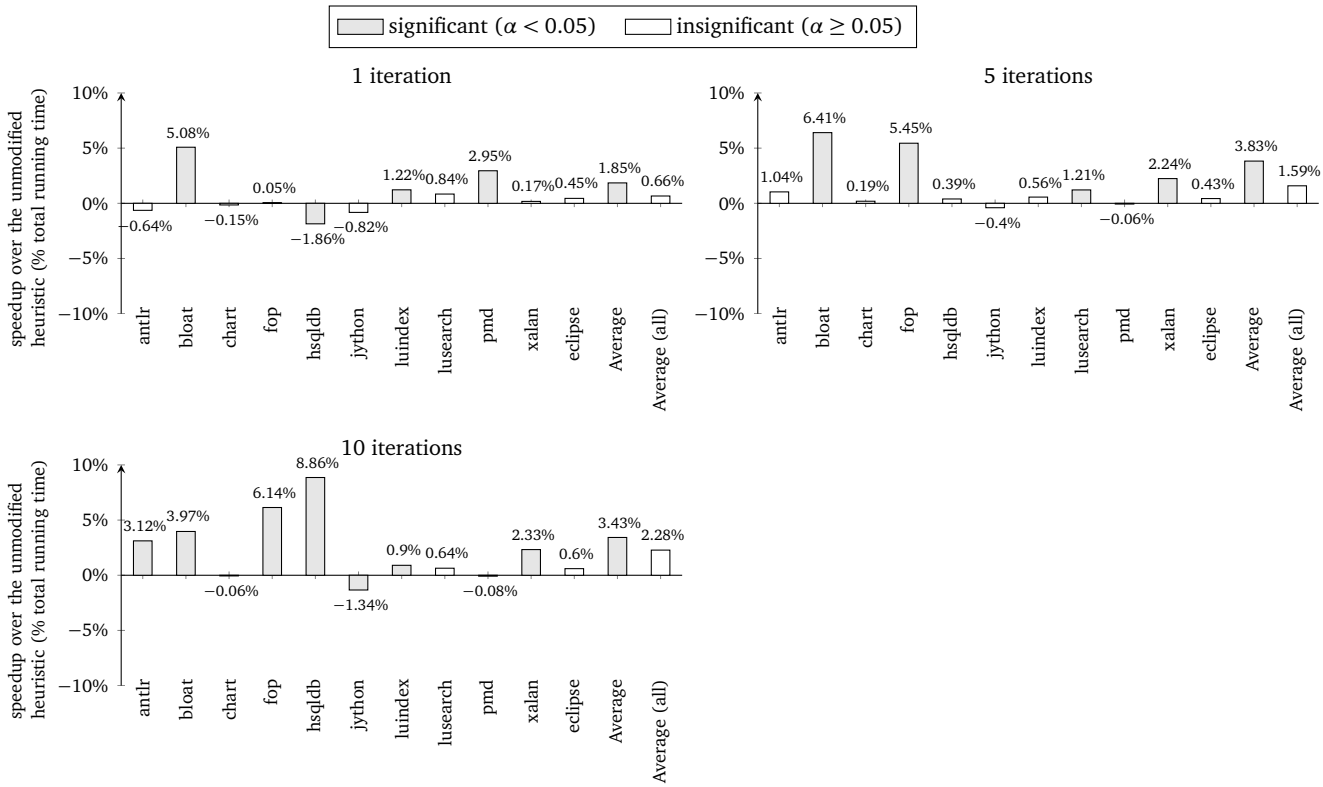

Figure 4.13: Speedup of the proposed heuristic with simulated long-running loads in a setting without replay compilation. The percentages shown represent the average speedup over fifteen invocations. The gray bars show benchmarks for which there is a statistically significant difference in total running time ($\alpha < 0.05$).

5 Related Work

The concept of performing inlining as a code optimization is probably as old as procedural programming itself. Nevertheless, inlining in static compilers as well as in just-in-time compilers is an active topic of contemporary research.

A main topic of inlining research is the cost-benefit tradeoff made when deciding whether to inline a particular call site. The state of the art is to use call-graph profiles to select the call sites for which inlining is particularly profitable [Gro+95]. Grove et al. implement profile-guided inlining based on offline profiles. They also experiment with profiling entire call chains as opposed to individual call edges. They find that the additional context information improves the pure program execution performance of the code.

Based on the finding that profiling longer call chains improves inlining decisions, Hazelwood and Grove [HG03] implement an online context-sensitive heuristic for Jikes RVM that achieves a significant reduction in compile time and code size with minimal performance degradation.

Bradel and Abdelrahman [BA05] extend the concept of profiling execution sequences from method call chains to traces of basic blocks, which are then used to drive inlining decisions. While they find that execution time of the optimized code is improved by about 10 %, the overhead of their profiling system is prohibitive at an overall execution-time increase of 139 %.

Zhao and Amaral [ZA04] observe that aggressive inlining is favorable for small benchmarks, while a more conservative setting should be used for larger benchmarks. They propose *adaptive inlining* as a profile-based technique that chooses appropriate inlining thresholds to adapt the inlining heuristic to different benchmark sizes.

Arnold et al. [Arn+00a] represent the cost-benefit tradeoff of inlining using the knapsack problem and provide a generalized approximation algorithm for the knapsack problem applied to inlining. They use this general algorithm to evaluate different heuristics based on offline profiles. The three heuristics are based on the static call graph, the node weights of the call graph, and the edge weights of the call graph, respectively. They evaluate the performance of their heuristics in a setting where compilation is performed ahead of time. They conclude that in their setting, performance is best for the heuristic based on edge weights.

Suganuma, Yasue, and Nakatani [SYN02] perform a similar study for online profile-directed inlining. They conclude that substantial performance improvements can be achieved by using online profile data to guide inlining decisions.

Even when profiles are used to guide inlining decisions, this still involves preset thresholds (that are possibly adapted dynamically) which influence the performance of the compiled code. Such thresholds are generally hand-tuned by compiler writers [CO05]. In recent years, the application of evolutionary algorithms and machine-learning algorithms to perform automatic tuning of compiler heuristics has been researched. Stephenson et al. [Ste+03] provide a general framework for constructing compiler heuristics based on machine-learning approaches. They use their framework to build heuristics for hyperblock formation, register allocation, and data prefetching.

Hoste, Georges, and Eeckhout [HGE10] use an evolutionary search algorithm to tune the parameters of Jikes RVM's adaptive optimization system. In a similar approach, Cavazos and O'Boyle [CO05] use a genetic algorithm to automatically tune the fixed inlining parameters of Jikes RVM. Their approach achieves total execution time speedups of up to 37 % for the Dacapo benchmark suite.

In addition to the selection of profitable call sites, there has also been considerable research on optimizing callee methods with regard to their parameters. Dean, Chambers, and Grove [DCG95] introduce *selective specialization*, which is a technique that compiles multiple versions of the same method for each possible argument class of the method according to the static information available in the caller context. Their contribution consists of performing specialization only in those cases where there is a considerable benefit to be gained.

Instead of trying to improve the cost-benefit tradeoff made when deciding whether to inline a particular call site, Chakrabarti and Liu [CL06] focus on the order in which call sites considered for inlining are processed by a static compiler. They use their compiler framework to drive *inline specialization*, which addresses the issue of selecting the most profitable, specialized version of a method for inlining into a particular call site.

Dean and Chambers [DC94] propose *inlining trials* as an approach to maximize the indirect benefits of inlining. An inlining trial consists of inlining a call site and recording the subsequent interprocedural optimizations in a summary database.

When deciding whether a similar call site should be inlined, the database is consulted to determine the resulting optimizations. The goal of this approach is very similar to that of the heuristic proposed in this thesis. Both approaches try to reap indirect benefits where they occur by inlining more aggressively in such cases. At the same time, inlining is performed more conservatively when no indirect benefits are likely to occur. What differentiates the work presented in this thesis from that of Dean and Chambers is the fact that the proposed heuristic uses readily-available profile information to efficiently determine whether a benefit is likely. Dean and Chambers perform inlining trials whenever a call site configuration was not previously recorded to determine if an indirect benefit occurs. While inlining trials require additional compilation to determine if indirect benefits are expected, the heuristic proposed in this thesis does not.

In this thesis, the indirect benefits of inlining caused by extant and precise arguments were examined. To achieve this, the peculiarities of inlining in the presence of virtual dispatch were described. These include the need for guard tests to ensure safety of dispatch when it cannot be guaranteed that a call site dispatches to a single target method. Then, the cases in which such guard tests can be omitted were discussed.

Based on this background, a formal notation was defined that allows reasoning about sequences of consecutive inlining operations. The propagation of extant and precise arguments into the callee context was examined using this notation. This led to a definition of when guards can be omitted as an indirect benefit of inlining. Based on this problem definition, a heuristic was proposed that uses readily-available call-graph profiles to predict when such benefits are likely to occur. These predictions were then used to restrict the assignment of size-estimate bonuses to cases in which an indirect benefit is possible according to the call-graph information. Two variants of this heuristic were also proposed. The first variant heuristic tries to predict indirect benefits that occur after multiple levels of the call hierarchy have been inlined into a single root method. The second variant focuses on the execution frequency of the call edge which suggests further inlining. If this call edge is executed frequently, the call edge weight of the initial inlining problem that enables the indirect benefit is increased to encourage inlining of that edge.

A benchmark-based evaluation was performed. In a first step, it was estimated how often precise and extant arguments are available at the call site. Then, the prediction quality of the proposed heuristic was quantified in terms of accuracy, precision and recall. The result of this quality evaluation is that about half of all instances of further inlining were predicted by the heuristic. At the same time, about 80 % of cases in which further inlining did not occur were correctly identified. The unmodified heuristic, the proposed heuristic, the variants of the proposed heuristic, and the zero heuristic that never assigns a size-estimate bonus for extant and precise arguments were compared in terms of performance. Overall, it was found that for some benchmarks, the zero heuristic saves compile time at the cost of degrading code quality and total execution time. The proposed heuristic saves a comparable amount of compile time, but has a much smaller negative impact on code quality for most benchmarks. The overall effect of the proposed heuristic is a significant improvement in total execution time for three of ten benchmarks and no significant change in overall performance for the other benchmarks when replay compilation is used. The variant heuristics achieve improvements for some benchmarks, but result in a degradation of performance for others. When replay compilation is not used, the proposed heuristic improves performance significantly by 1 % to 5 % for three benchmarks, while a significant performance reduction of 2 % is observed for a single benchmark.

The results of the evaluation show that it is feasible to use live call-graph profiles to form predictions about further inlining. It was shown that about half of all instances of further inlining were correctly predicted. The risk of predicting further inlining when none occurs is lower than 20 % for most benchmarks. Using the predictions to adjust the size-estimate bonuses can improve overall execution performance as shown by the performance measurements. Compared to the unmodified inlining heuristic, the proposed heuristic discourages the inlining of call sites with extant or precise arguments when further inlining of methods invoked on these arguments is less likely to occur. This more selective inlining causes a reduction in compile time that improves overall execution time for some benchmarks.

6.1 Future Work

While the proposed heuristic improves overall execution performance profitably for some benchmarks by reducing compile time, an improvement in code quality is not achieved. This is due to the fact that the proposed heuristic only prevents the inline oracle from providing additional incentives for inlining call sites with extant or precise arguments when no further inlining is expected. When further inlining is expected, the proposed heuristic assigns the same size-estimate bonuses as the unmodified heuristic in order to provide an incentive for inlining the call site.

As an approach to exploit more opportunities for indirect benefits caused by extant and precise arguments, the conductive-call-graph heuristic was proposed. This heuristic increases the incentive for inlining a call site with precise or extant arguments when it is expected that these arguments cause indirect benefits for inlining a hot call edge. This approach leads to an improvement in code quality for

some benchmarks, and in some cases even improves total execution performance when compared to the originally-proposed heuristic. However, for other benchmarks this approach leads to an increase in compile time that degrades overall performance. For some benchmarks, an actual degradation of code quality was observed. The isolated positive results suggests that there is some potential for improving overall performance by incentivizing inlining of call sites when indirect benefits for further inlining are expected. It is possible that this approach can be tuned in such a away that its overall performance gain exceeds that of the originally-proposed heuristic. However, this requires an understanding of the indirect benefit of guardless inlining in terms of execution speed gained. Currently, such indirect benefits are estimated in terms of code size reductions that only include improved execution performance implicitly.

In order to better exploit the indirect benefits caused by precise and extant arguments, an adaptive approach may also be useful. As described in Section 2.2, Jikes RVM uses an adaptive inlining organizer that monitors the call-graph profile for frequently-executed call edges. When such a hot edge is found, it is marked for future inlining. Additionally, the adaptive optimization system (cf. Section 2.4.1) is encouraged to recompile the caller in order to cause inlining of the hot edge. This suggests another approach for exploiting the indirect benefits of inlining.

As a prerequisite, the compiler would have to be modified to record all precise and extant arguments of previously-compiled call sites. Once this information is available, a modified adaptive inlining organizer could analyze the recorded call-site information and the hot edge by using the definitions of precise-induced and extant-induced edges (cf. Definition 3.5 and Definition 3.6). This may reveal that inlining the caller of the hot edge into one of its parent callers may make precise or extant information for the receiver of the call edge available. This information may then be used to inline the hot edge guardlessly. In this case, it is probably profitable to also inline the caller of the hot edge into its parent caller instead of only inlining the hot edge. This may cause extant or precise information to be propagated. However, this also requires the recompilation of the corresponding parent caller, which incurs an additional cost.

The outlined approach involves the creation of additional data structures that need to be maintained by the compiler and kept in memory, which represents an additional runtime overhead. However, implementing such an approach is desirable for two reasons. Firstly, the resulting adaptive heuristic would exploit indirect benefits in a proactive manner instead of restricting inlining when no benefits are expected. Intuition suggests that this may result in improved code quality. Secondly, the action of recompiling the parent caller is only taken if an edge is actually observed that suggests that action to be profitable. This means that the suggested adaptive optimization only behaves differently from the normal inlining heuristic in situations where there is enough call-graph data to support its action. This is an advantage over the proposed heuristic, for which missing call-graph edges cause opportunities for indirect benefits to be missed that are exploited by the unmodified heuristic.

Implementing the outlined approach would require a way of determining when it is profitable to recompile the parent caller to enable guardless inlining. The heuristic that was implemented as part of this thesis achieves its performance effect by manipulating the cost-benefit tradeoff of the inline oracle. The inline oracle of Jikes RVM makes this tradeoff on the basis of a size estimate of the callee that is being considered for inlining. The execution time benefit of inlining is never made explicit in this kind of tradeoff. In order to implement the outlined adaptive inlining strategy, this is insufficient. The outlined strategy would recompile additional methods to enable an indirect benefit. The adaptive optimization system of Jikes RVM [Arn+00b] decides whether to recompile a particular method by weighing the expected time required for the recompilation against the expected speedup of the recompiled method over the current compiled version of the method. This computation is based on empirical measurements of the differences between the compilers (cf. Section 2.4.1) and allows a precise tradeoff in terms of explicit cost and benefit to be made. When the reason for recompilation is a presumed indirect benefit of guardlessly inlining a hot call edge, the benefit that is caused by the recompilation is not easily expressed in terms of saved execution time. Therefore, a prerequisite of implementing the adaptive strategy outlined above is to find a cost-benefit equation for inlining in which the indirect benefits of inlining are expressed explicitly in terms of execution time. Since developing such a cost-benefit equation represents a large research topic of its own, implementing the outlined strategy was not attempted as part of this thesis.

Bibliography

[AG05] Matthew Arnold and David Grove. “Collecting and Exploiting High-Accuracy Call Graph Profiles in Virtual Machines.” In: *CGO ’05: Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2005, pp. 51–62.

[AR02] Matthew Arnold and Barbara G. Ryder. “Thin Guards: A Simple and Effective Technique for Reducing the Penalty of Dynamic Class Loading.” In: *ECOOP ’02: Proceedings of the 16th European Conference on Object-Oriented Programming*. Springer Verlag, 2002, pp. 498–524.

[ASU07] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. 2nd ed. Pearson Education, 2007.

[Arn+00a] Matthew Arnold, Stephen Fink, Vivek Sarkar, and Peter F. Sweeney. “A Comparative Study of Static and Profile-Based Heuristics for Inlining.” In: *DYNAMO ’00: Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*. Vol. 35. ACM Press, 2000, pp. 52–64.

[Arn+00b] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. “Adaptive Optimization in the Jalapeño JVM.” In: *OOPSLA ’00: Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, 2000, pp. 47–65.

[Ayc03] John Aycock. “A Brief History of Just-In-Time.” In: *ACM Computing Surveys*. Vol. 35. ACM Press, 2003, pp. 97–113.

[BA05] Borys J. Bradel and Tarek S. Abdelrahman. “The Use of Traces for Inlining in Java Programs.” In: *Languages and Compilers for High Performance Computing*. Ed. by Rudolf Eigenmann, Zhiyuan Li, and Samuel Midkiff. Vol. 3602. Lecture Notes in Computer Science. Springer Verlag, 2005, pp. 922–922.

[Bla+06] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, 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.” In: *OOPSLA ’06: Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, 2006, pp. 169–190.

[Bla+08] Stephen M. Blackburn, Kathryn S. McKinley, Robin Garner, Chris Hoffmann, Asjad M. Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovik, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. “Wake Up and Smell the Coffee: Evaluation Methodology for the 21st Century.” In: *Communications of the ACM* 51 (8 2008), pp. 83–89.

[CL06] Dhruva R. Chakrabarti and Shin-Ming Liu. “Inline Analysis: Beyond Selection Heuristics.” In: *CGO ’06: Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2006, pp. 221–232.

[CLS00] Michal Cierniak, Guei-Yuan Lueh, and James M. Stichnoth. “Practicing JUDO: Java Under Dynamic Optimizations.” In: *PLDI ’00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. ACM Press, 2000, pp. 13–26.

[CO05] John Cavazos and Michael F. P. O’Boyle. “Automatic Tuning of Inlining Heuristics.” In: *SC ’05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society, 2005, pp. 14–.

[DA99] David Detlefs and Ole Agesen. “Inlining of Virtual Methods.” In: *ECOOP ’99: Proceedings of the 13th European Conference on Object-Oriented Programming*. Springer Verlag, 1999, pp. 258–278.

[DC94] Jeffrey Dean and Craig Chambers. “Towards Better Inlining Decisions Using Inlining Trials.” In: *LFP ’94: Proceedings of the 1994 ACM Conference on LISP and Functional Programming*. ACM Press, 1994, pp. 273–282.

[DCG95] Jeffrey Dean, Craig Chambers, and David Grove. “Selective Specialization for Object-Oriented Languages.” In: *PLDI ’95: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*. ACM Press, 1995, pp. 93–102.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. “Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis.” In: *ECOOP ’95: Proceedings of the 9th European Conference on Object-Oriented Programming*. Ed. by Mario Tokoro and Remo Pareschi. Vol. 952. Lecture Notes in Computer Science. Springer Verlag, 1995, pp. 77–101.

[GEB08] Andy Georges, Lieven Eeckhout, and Dries Buytaert. “Java Performance Evaluation through Rigorous Replay Compilation.” In: *OOPSLA ’08: Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, 2008, pp. 367–384.

[Gro+95] David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers. “Profile-Guided Receiver Class Prediction.” In: *OOPSLA ’95: Proceedings of the 10th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, 1995, pp. 108–123.

[HCU92] Urs Hölzle, Craig Chambers, and David Ungar. “Debugging Optimized Code with Dynamic Deoptimization.” In: *PLDI ’92: Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*. ACM Press, 1992, pp. 32–43.

[HG03] Kim Hazelwood and David Grove. “Adaptive Online Context-Sensitive Inlining.” In: *CGO ’03: Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2003, pp. 253–264.

[HGE10] Kenneth Hoste, Andy Georges, and Lieven Eeckhout. “Automated Just-In-Time Compiler Tuning.” In: *CGO ’10: Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM Press, 2010, pp. 62–72.

[MRS08] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. 1st ed. Cambridge University Press, July 2008.

[SYN02] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. “An Empirical Study of Method Inlining for a Java Just-In-Time Compiler.” In: *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium*. San Francisco, California, USA: USENIX, 2002.

[Spe] *SPECjvm98 Documentation, Release 1.01*. URL: `ttp://www.spec.org/osg/jvm98/jvm98/doc/index.html` (visited on 05/29/2011).

[Ste+03] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O’Reilly. “Meta Optimization: Improving Compiler Heuristics with Machine Learning.” In: *PLDI ’03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. ACM Press, 2003, pp. 77–90.

[ZA04] Peng Zhao and Jóse Amaral. “To Inline or Not to Inline? Enhanced Inlining Decisions.” In: *Languages and Compilers for Parallel Computing*. Ed. by Lawrence Rauchwerger. Vol. 2958. Lecture Notes in Computer Science. Springer Verlag, 2004, pp. 405–419.

A Supplementary Data

This appendix contains the detailed results of measuring performance for the proposed heuristic using the simulated long-running loads. The measurements presented here were performed without using replay compilation.

Figure A.1 shows the aggregate compile time and program time for running each benchmark for a total duration of five iterations. The plots for the benchmarks bloat, chart, fop, luindex, lusearch, pmd, and eclipse show better compile time for the proposed heuristic. For the benchmarks bloat, chart, luindex, lusearch, and eclipse, some increase of program time is visible. For xalan, there is an outlier visible that shows a strong reduction in compile time and an equivalent increase in program time.

Figure A.2 shows the aggregate compile time and program time for running ten iterations of each benchmark. There is a clear separation in compile time for bloat, chart, fop, hsqldb, luindex, lusearch, pmd, xalan, and eclipse visible. For hsqldb, jython, lusearch, and xalan, a degradation in program time is visible for the proposed heuristic.

Both plots show that the long-running loads make the difference of the proposed heuristic and the unmodified heuristic more pronounced, especially with regard to compile time. The average effect of the proposed heuristic in terms of total running time for this setting is shown in Figure 4.13 (cf. Section 4.9).

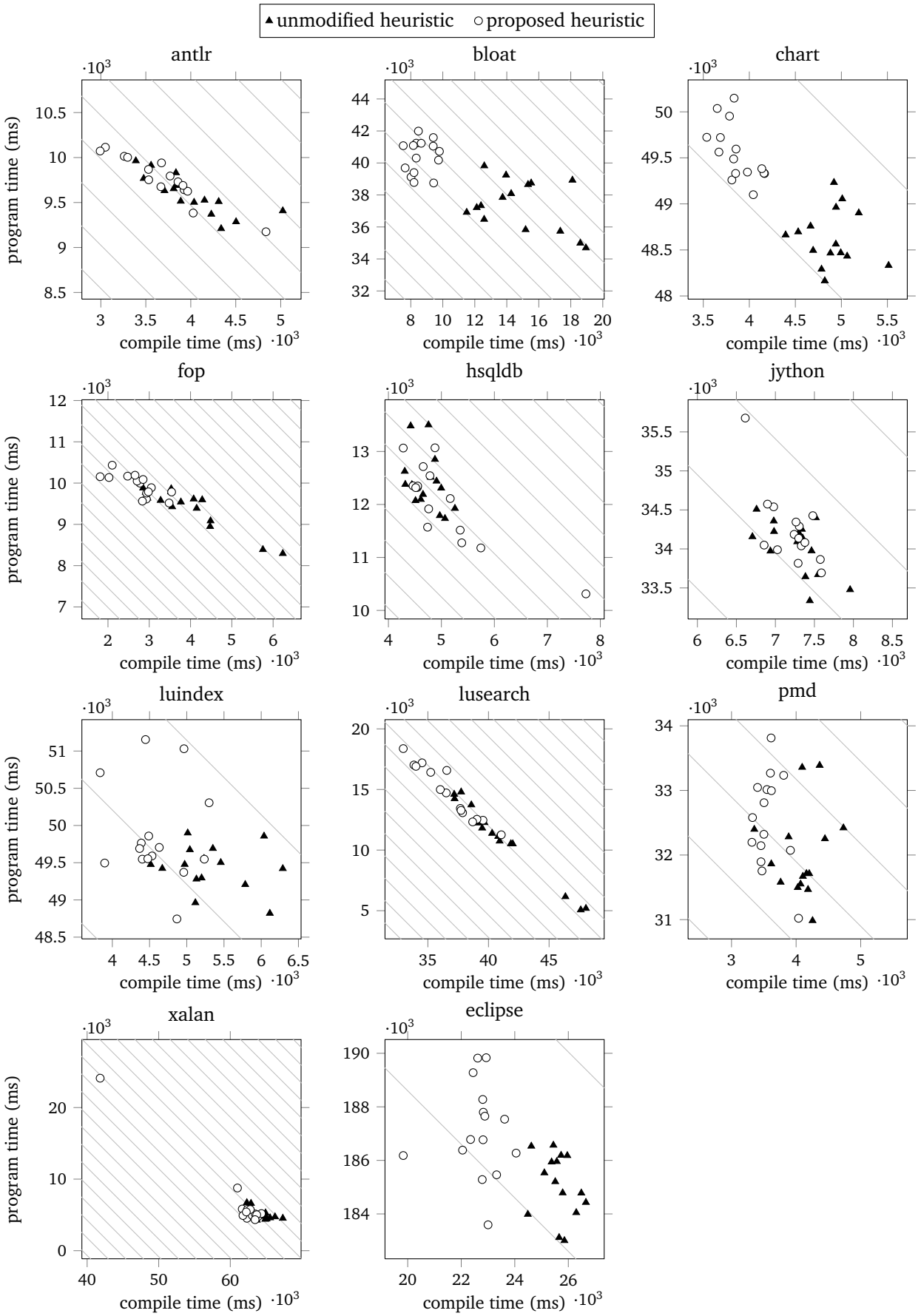

Figure A.1: The effect of the proposed heuristic on compile time and pure program time in a setting without replay compilation. Each mark represents the result of a single invocation of five iterations of the benchmark. The diagonal lines visualize total running time. Two marks on the same line have the same total running time. The distance between two lines represents 2.5% of the average total running time for the unmodified heuristic.

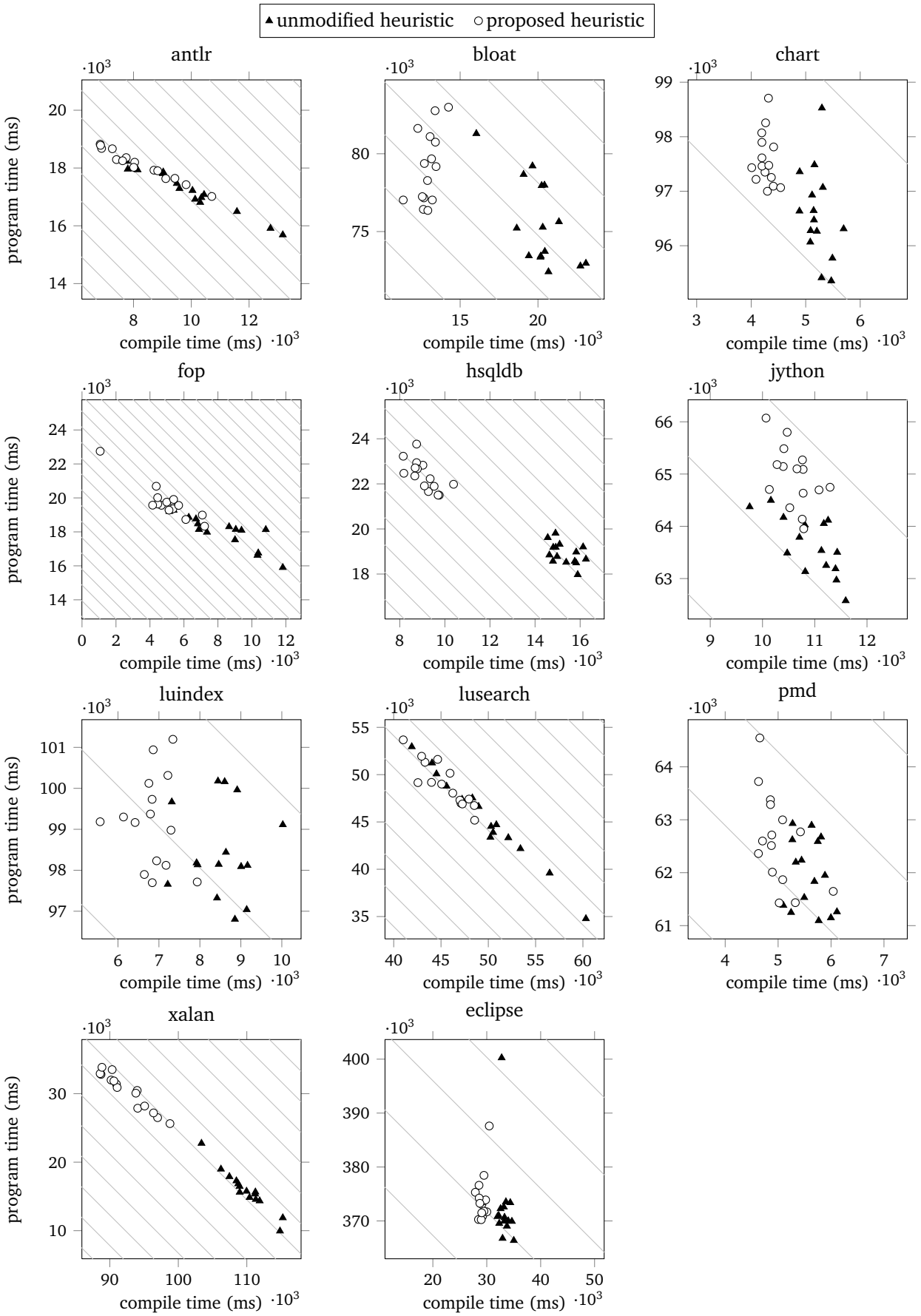

Figure A.2: The effect of the proposed heuristic on compile time and pure program time in a setting without replay compilation. Each mark represents the result of a single invocation of ten iterations of the benchmark. The diagonal lines visualize total running time. Two marks on the same line have the same total running time. The distance between two lines represents 2.5% of the average total running time for the unmodified heuristic.

B List of Figures and Tables

List of Figures

List of Tables

C Evaluation Tools

In the progress of this work, a number of shell scripts and command-line tools were created that collect, summarize, and visualize data.

C.1 Setup

The code and tools produced as a part of this thesis are delivered in the form of a copy of the Mercurial[1] repository for Jikes RVM. The branch `measurements` contains all measurement tools. In order to use the measurement scripts, create a directory (identified by the placeholder <root> here), where all results and temporary data will reside. Copy the delivered repository into <root> as a subdirectory. The placeholder <repository> will be used as the name for the absolute path of the repository. Update the mercurial repository to the `measurements` branch. Then copy the file <repository>/settings.example to <repository>/settings. Edit the settings file according to your setup:

```
BASEDIR="<root>"
REPOSITORY="<repository>"
ARCH=ia32-linux # change to reflect your architecture according to Jikes RVM
EXECUTABLE="dist/production_${ARCH}/rvm"
VM_OPTS="-Xmx512M"
BENCHMARKS="antlr bloat chart fop hsqldb jython luindex lusearch pmd xalan"
BENCHMARK_JAR="${REPOSITORY}/components/dacapo/2006-10-MR2/dacapo.jar"
```

To download the Dacapo benchmark suite, execute the following command inside <repository>:

```
ant -f build/components/dacapo.xml
```

C.2 Performance Measurements

In order to measure performance for the different evaluated heuristics, a number of measurement scripts were created that are located at the root of <repository>. Some measurement scripts use replay compilation and require a set of compilation plans, others do not. All performance measurement scripts are supplied one or more mercurial branch names of Jikes RVM versions. The first one of them serves as a baseline against which the other branches are compared. This is normally the `trunk`, but the placeholder <baseline> will be used here. The placeholder <branch>+ is a space-separated list of branch names.

All performance measurements should be performed in single-user mode, if available. However, an X11 Server is required if the benchmark `chart` is selected in the settings file. For this, Xvfb[2] has proved useful as a mock X11 Server. To use Xvfb with the provided measurement scripts, execute them using the utility program `xvfb-run`.

C.2.1 Performance Measurements with Replay Compilation

As a first step, compilation plans need to be created. For this, the system should be running in single-user mode. Compilation plans only need to be generated once: They can and should be reused for every measurement on the same machine. To generate a set of compilation plans called <planset> with <plans> compilation plans for each benchmark and <iterations> iterations each, run the following command line inside <repository>:

```
./generate-compilationplans.sh trunk <planset> <plans> <iterations>
```

When the measurement has completed, the set of compilation plans will be located at <root>/benchmarks/plansets/<planset>. Using this planset, a replay-based performance comparison can be invoked like this:

[1] http://mercurial.selenic.com/

[2] http://linux.die.net/man/1/xvfb

```
./compare-revisions.sh <planset> <baseline> <branch>+
```

The result is written to the directory <root>/benchmarks/<identifier>, where <identifier> is a string that consists of <baseline> and <branch>+ concatenated with underscores. The main output file is called output_<iterations>it_<date>.txt.bz2. In addition to this, the analysis tool as well as pdflatex will be run to generate summary statistics and plot them into a pdf file called result_<iterations>it_<date>.pdf.

In order to re-generate the summary statistics by hand, run the following command inside <repository>:

```
./build-analysis.sh <outfile> <planset>
```

The placeholder <outfile> is the compressed measurement result file (.txt.bz2) generated by the above measurement script, while <planset> is the name of the set of compilation plans that was used to perform the measurement.

There is a variant of the measurement script that compares revisions using different command-line arguments supplied to the measured rvm executable. This measurement script is invoked as follows:

```
./compare-configurations.sh <planset> <config> <baseline> <branch>+
```

The placeholder <config> is a directory that contains a number of single-line text files. Each of these text files contains one or more command-line switches. When the script is executed, every supplied branch except <baseline> will be executed for every benchmark, compilation plan and configuration. This measurement script can for example be used to tune inlining parameters by brute force.

C.2.2 Non-Replay Measurements

To compare the performance of different branches without using replay compilation, invoke the following command:

```
./compare-revisions-nonreplay.sh <invocations> <iterations> \
  <baseline> <branch>+
```

The placeholder <invocations> is the number of invocations to measure for each benchmark, while <iterations> is the number of iterations to perform for each invocation. The result is written into the directory <root>/benchmarks/nonreplay_<identifier>, where <identifier> is a string that consists of <baseline> and <branch>+ concatenated with underscores. The same kind of output is generated as for the replay measurement. To re-generate the summary statistics, invoke:

```
java -cp analyser.jar CompareRevisionsNonReplay <outfile> \
  <repository>/plots/compare-revisions-template/comparison-nonreplay.tex
```

The placeholder <outfile> is the uncompressed output file of the measurement script.

C.2.3 File Format

The files generated by the measurement scripts are a semicolon-separated table with the following columns:

```
benchmarkId; benchmark; plan; planIterations; revision; time1; time2; time3; \
baseCompileTime; baseMCKB; optCompileTime; optMCKB
```

There is one numbered time column for each iteration that was measured.

C.3 Recording and Analyzing Inlining Decisions

To record the inlining decisions of a set of branches, run the following command:

```
./compare-inlining-revisions.sh <planset> <baseline> <branch>+
```

The branches are cloned from the repository and patched using print_inline_sequence.patch and print_size_and_limits.patch which output the log information that is used by the analyzer in the next step. To record the predictions of the proposed heuristic, either the branch inlining-simple-callgraph-verbose or inlining-heuristic-accuracy must be used.

Inlining decisions are recorded separately for all plans. The results of the measurement can be found in the directory <root>/benchmarks/inline_decisions_<identifier>, where <identifier> is a string that consists of <baseline> and <branch>+ concatenated with underscores.

C.3.1 The Inlining Report Analyzer

The inlining report analyzer works on the output of the script compare-inlining-revisions.sh. It gathers statistics about individual inlining decisions that are then summarized into multiple text files. Additionally, LaTeX[3] documents are generated that compile to visualizations of the data. In order to generate general statistics about inlining decisions, the program is invoked as follows:

```
java -Xmx1024M -cp analyser.jar InliningReportAnalyser <outdir> \
  <repository>/plots/compare-revisions-template/inline-decisions.tex\
<baseline>
```

In this command-line, <outdir> is the output directory created by the measurement script. The tool creates a number of text files that summarize inlining decisions. Two LaTeX files, inline-decisions.tex and inline-sizes.tex are created that show plots of the results. In the subdirectory decisions, text files are created that compare the individual decisions of the compared branches.

C.3.2 Computing the Accuracy Statistic

To compute the accuracy statistic, the inlining report analyzer is used. Accuracy can only be computed for the branches inlining-simple-callgraph-verbose and inlining-heuristic-accuracy. The default is to use the latter since it only records the predictions of the proposed heuristic but assigns the bonuses like the unmodified heuristic. Since computing $\mathscr{P}$ (cf. Section 3.1.6) requires analyzing the class hierarchy of the program, computing the accuracy requires the executed benchmark (e.g. Dacapo) to be on the classpath. To compute accuracy, invoke the analyzer as follows:

```
java -Xmx1024M -cp analyser.jar:dacapo.jar InliningReportAnalyser \
  -accuracy <outdir> \
  <repository>/plots/compare-revisions-template/inline-decisions.tex \
  <baseline>
```

This will output all statistics that are created by the command line in Section C.3.1 plus the accuracy statistics.

C.3.3 File Format

The output resulting from the inlining report patches looks like the example below:

```
InlineDecision [< SystemAppCL, Ldacapo/TeeOutputStream; >.write (I)V|2|\
  <BootstrapCL, Ljava/io/FilterOutputStream; >.write (I)V|false]
InlineSequence [< SystemAppCL, Ldacapo/TeeOutputStream; >.write (I)V&-1|]
BONUS_CHECK dacapo.TeeOutputStream
 - has edge to java.io.PrintStream
BONUS_NO: dacapo.TeeOutputStream
        YES: trivial guardless inline
```

The first line shows the call site that is evaluated for inlining. The first method descriptor inside the brackets is the caller, after the first pipe symbol the bytecode index that identifies the call site inside the caller is given ("2" in this example). After another pipe symbol, the method descriptor of the callee is shown. After that, a boolean false signifies that the call site to inline is not an invokespecial or invokestatic call site. The next line shows the inline sequence. Here, only the root method is shown since the method that is considered for inlining is at the first level of the inline sequence. The next three lines show the *decision details* that include the verbose output of the heuristic that checks outgoing call-graph edges that suggest a call on an argument. The last line shows the reached decision with the reason for the decision.

[3] www.latex-project.org