Next In Line, Please!

Exploiting the Indirect Benefits of Inlining by Accurately Predicting Further Inlining

Andreas Sewe Jannik Jochem Mira Mezini

Technische Universität Darmstadt

{sewe, jochem, mezini}@st.informatik.tu-darmstadt.de

Abstract

Inlining is an important optimization that can lead to significant runtime improvements. When deciding whether or not to inline a method call, a virtual machine has to weigh an increase in compile time against the expected decrease in program time. To estimate the latter, however, state-of-the-art heuristics only use information local to the call-site in question. But inlining is a powerful enabling optimization; by eliminating the actual call it not only offers an obvious direct benefit but also indirect benefits, as information about the method's arguments is propagated from caller to callee. One such indirect benefit is the elimination of guards in case the callee inlines a method called on one of its arguments. In this paper, we show how to enhance an inlining heuristic by accurately predicting where this further inlining occurs—and where not. To do so, we only use information readily available to many virtual machines: the program's dynamic call graph. An implementation based on Jikes RVM demonstrates that this information can be used to successfully exploit inlining's indirect benefits while at the same time reducing compilation effort.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers, Optimization

General Terms Performance

Keywords Inlining, indirect benefits, guard elimination

1. Introduction

Well-designed object-oriented programs typically consist of a large number of small methods. But such a design comes at a price: Every single method call causes overhead which, taken together, can cause a significant increase in runtime. Modern compilers therefore use method inlining, which works by replacing a call-site within a caller method with the body of the respective callee method. This not only completely removes the overhead of the method call, but may also enable further optimizations, as static information is now propagated across method boundaries, i.e., from caller to callee.

The use of method inlining is not free, though. Both code size and compile time can increase when numerous call-sites are replaced by a method's entire body. The compiler thus has to carefully weigh the costs of inlining against its benefits. This cost-benefit trade-off is of particular relevance to modern virtual machines where a just-in-time compiler performs the optimization at runtime. To this effect, compilers employ a so-called inlining heuristic that tries to predict both the cost and the benefits of an inlining decision at compile time.

State-of-the-art inlining heuristics, however, only use crude approximations of the indirect benefits of inlining. One important such benefit is the inlining of further methods the callee calls on its arguments. Such situations are particularly common when a program uses the Command pattern either directly or indirectly, e.g., when a compiler resorts to its use to emulate first-class functions:

```
Collections.sort(list, new Comparator() {
  int compare(Object lhs, Object rhs) {
    ...
  }
});
```

The compare method, which will likely be called numerous times during sorting, can only be inlined (without guards) if information about the precise comparator used is propagated from caller to callee, i.e., into the body of sort; this happens only if the marked call-site is inlined. State-of-the-art heuristics therefore encourage inlining when information about the callee's arguments, e.g., their precise types, are known. They do so, however, indiscriminately, i.e., regardless of whether or not a method is actually called on the argument in question, in a desire to avoid a detailed static analysis of the callee's code.

In this paper we side-step this issue by proposing an inlining heuristic which instead uses the dynamic call graph to predict whether such a method call will actually happen. If no matching edge is found in the call graph, propagating information about the argument from caller to callee is unlikely to produce the desired indirect benefit; inlining for this reason alone should thus be discouraged. No static analysis is needed to determine this. This heuristic reduces both compile time and code size with negligible effects on code quality.

The contributions of this paper are threefold:

1. We introduce and motivate the prediction problem of further inlining.
2. We propose an enhanced inlining heuristic that accurately predicts further inlining and guides inlining accordingly.
3. We rigorously evaluate this heuristic on Jikes RVM [1] using the DaCapo benchmark suite [5].

This paper is structured as follows: Section 2 provides background material on the inlining optimization in general and its implementation in Jikes RVM in particular. Section 3 describes our proposed heuristic, which is then evaluated in Section 4. Section 5 discusses related work, before Section 6 concludes.

2. Background

In this section, we first provide some background on the inlining optimization together with a concise notation to reason about how detailed information about the reference arguments can enable guardless inlining. We then give an overview of the inlining heuristic used by Jikes RVM [1].

2.1 The Inlining Optimization

Inlining is the process of replacing a method's call-site with the body of said method. This optimization avoids creating a stack frame for the callee method and transferring control from caller to callee and back. Furthermore, in object-oriented languages, where calls are dynamically dispatched, the cost of dispatching can often be avoided by exploiting the fact that in practice many call-sites are monomorphic rather than polymorphic [15].

Guarded and Guardless Inlining While many call-sites are indeed monomorphic, the compiler is not always able to prove this fact statically, e.g., by applying class-hierarchy analysis [11]. This is particularly true in modern, managed languages like Java, where dynamic class loading can grow the class-hierarchy at runtime.

Still, it may be profitable to inline the most likely callee, even when the compiler is unable to prove that the call-site in question is monomorphic. In such cases, compilers resort to guarded inlining [17]; they insert so-called guards which test whether some precondition is met before executing the inlined callee. If the guard test fails, dynamic dispatch is performed as a fall-back. In all cases, however, testing a precondition incurs some minor overhead.

While techniques exist to reduce the overhead of guarded inlining [2, 12], being able to inline a virtual method without any guard is better still. In Java, this is mainly possible in two cases:[1] when the receiver's precise type is statically known and when the receiver object already pre-exists the call [12].

Notation In order to reason about these cases, we next introduce notation to discuss the decision the compiler has to make: "Should a callee B.n() be inlined into a caller A.m()?"

Notation 2.1. $\mathsf{A.m()} \overset{?}{\leftarrowtail} \mathsf{B.n()}$

Note that B above is the actual target of the call. Should the compiler decide to inline such a call, with or without a guard test, an inline sequence is begun. The following notation describes such sequences: "C.o() is inlined into B.n() which is in turn inlined into the root method A.m()."

Notation 2.2. $\mathsf{A.m()} \leftarrowtail \mathsf{B.n()} \leftarrowtail \mathsf{C.o()}$

Such sequences are of particular interest if inlining propagates information about the method's arguments from caller to callee. We thus extend our notation to include arguments: "Should a callee B.n() be inlined into a caller A.m() if it has an argument of static type C?"

Notation 2.3. $\mathsf{A.m()} \overset{?}{\leftarrowtail} \mathsf{B.n(C)}$

In the above, B.n(C) has just one argument. It may of course also have several, scalars and arrays alike, of either primitive or reference type. When making inlining decisions, however, it is sufficient to consider the arguments one by one. Moreover, as dispatch in Java is based on a single object only, guard elimination requires information about just the scalar reference arguments; both array and primitives can be ignored. Also, the implicit this argument can, for the purpose of this discussion, be treated like any other argument.

```
class A {
  void m() {
    B b = ...
    C c = new D();
    b.n(c); // Precise type of argument is D.
  }
}

class D extends C { ... }
```

Figure 1. The type of c is known precisely in A.m(C); inlining the marked call-site propagates this information into B.n(C).

```
class A {
  void m(C c) {
    B b = ...
    b.n(c); // Argument exists before call to A.m(C).
  }
}
```

Figure 2. The value of c pre-exists the invocation of A.m(C); inlining the marked call-site propagates this information into B.n(C).

Now, it may be the case that the compiler has additional information about an argument, which can be exploited once it has been propagated from caller to callee. One such piece of information is the precise type of an argument at the call-site in question. In the situation depicted in Figure 1, the caller A.m() propagates information about the precise type of an argument into the callee B.n(C), which may allow the compiler to prove call-sites within the callee monomorphic—if the caller is inlined into the callee.

Notation 2.4. $\mathsf{A.m()} \overset{?}{\leftarrowtail} \mathsf{B.n(D^\sharp <: C)}$

Another additional information is that the argument pre-exists the root method. This pre-existence guarantees that the argument's class has been fully loaded before the root method was called [12]. If a method is called on the argument and the respective call-site is currently provably monomorphic, then no guard is necessary to inline the call. While it is still possible that the call-site may become potentially polymorphic later on, simply recompiling the calling method is enough to ensure that dynamic dispatch is handled correctly in all cases; neither on-stack replacement [13] nor code patching [8] are necessary. In the situation depicted in Figure 2, the caller A.m(C) propagates the information that the argument that pre-exists its own invocation into the callee B.n(C)—if the latter is inlined into the former.

Notation 2.5. $\mathsf{A.m(C^\flat)} \overset{?}{\leftarrowtail} \mathsf{B.n(C^\flat)}$

In either case, information about the argument of B.n(C) is propagated during inlining into A.m(...). If B.n(C) now calls a method on the argument in question, guardless inlining becomes possible. In fact, Jikes RVM's inlining heuristic already handles both of these cases: precise and extant arguments.[2]

2.2 The Inlining Heuristic of Jikes RVM

Jikes RVM [1] has two compilers: an extremely fast baseline compiler and a much slower optimizing compiler. Only the latter performs inlining, guided by a so-called inline oracle [16].

[1] Another case is when the receiver's precise type is unknown, but final classes and methods make it possible to deduce a single target method.

[2] In Jikes RVM, pre-existing arguments are called extant; both terms are used interchangeably in this paper.

Information	Reduction
Reference argument of precise type	15 %
Reference argument pre-exists method call	5 %
Non-null object constant	10 %
null constant	10 %
Integer constant	5 %
Array argument of precise type	5 %
No aastore check required	2 %

Table 1. Estimated reductions in a method's size if additional information is available about an argument.

The Inline Oracle This oracle bases its inlining decisions (no inlining, guarded inlining, guardless inlining) on both static and dynamic information available in the current compilation context. Static information includes a method summary of the callee including an estimate of its size as well as the types of both the call's receiver object and the call's arguments, approximated statically through intra-procedural dataflow analysis. Dynamic information includes the hotness of the call-edge in question. Furthermore, the inline oracle is influenced by compiler settings; more than 20 options exist to fine-tune the cost-benefit analysis performed by the oracle. The actual decision is reached in a five-step process:

1. Reject certain callees which should not (@NoInline-annotated methods) or cannot (native methods) be inlined.
2. Accept trivial callees, i.e., methods of negligible size.
3. Use the dynamic call graph to identify the dynamic targets of the potentially polymorphic call-site.
4. For each dynamic target, decide whether inlining is desirable.
5. Choose appropriate guards for each desirable target.

Of these five steps, the last three are performed only at higher optimization levels (O1–O2). The most complex step thereby is the fourth, in which the oracle performs a cost-benefit analysis for each dynamic target.

Cost-benefit Analysis To estimate the cost, the oracle estimates both the size of the callee as well as the size of any guards necessary. If the inline oracle has additional information about the callee's arguments, it will reduce its estimate of the callee's size accordingly. Each argument is considered separately, the respective reductions are added together and finally applied to the size estimate. The maximum reduction is limited to 40%; thus, the final size estimate is at least 60% of the original estimate. Table 1 shows the (per-argument) size reductions attributed by default to various information about the arguments.

If the callee's size, after all reductions have been applied, is below some configurable threshold, the callee is inlined, a strategy remarkably similar to that of the Self-93 compiler [17]. At first glance, the oracle hereby seems to consider the cost of inlining only. But benefits are also, albeit indirectly, factored into the cost-benefit trade-off: by reducing the cost of inlining. Since the cost-benefit trade-off is a simple inequation, reducing the cost has the same effect on the final inlining decision as increasing the benefit.

Prevalence of Precise and Extant Arguments Table 2 shows how often the precise type of an argument or its pre-existence are known to the optimizing compiler of Jikes RVM, i.e., how prevalant precise and extent arguments are in practice. The first pair of columns shows the percentage of non-receiver scalar reference arguments that are extant or precise, respectively. The next pair of columns shows the percentage of extant or precise receivers, which

	Non-receiver Arguments [%]		Receivers Arguments [%]		All Arguments [%]	
Benchmark	Extant	Precise	Extant	Precise	Extant	Precise
antlr	21.11	39.46	26.41	38.88	25.09	38.95
bloat	19.90	44.87	7.87	69.12	12.78	59.22
chart	8.93	37.55	6.77	70.71	7.75	55.76
fop	43.08	22.56	34.91	34.36	39.29	28.01
hsqldb	21.74	13.64	15.10	29.08	18.56	21.00
jython	43.75	14.83	27.42	46.78	36.89	28.27
luindex	6.83	61.12	38.39	22.81	31.64	30.86
lusearch	22.36	40.39	60.60	22.05	51.82	26.26
pmd	11.29	27.49	17.89	43.42	14.82	35.98
xalan	35.50	11.88	27.31	25.83	30.71	20.05
Average	23.45	31.38	26.27	40.30	26.94	34.44

Table 2. Prevalence of precise and extant scalar reference arguments in the Dacapo benchmark suite [5], eclipse excluded (Arithmetic mean of 10 compilation plans / benchmark).

are by definition scalars of reference type. The last two columns show the respective percentages for all arguments, whether explicit or implicit, i.e., the receiver. As can be seen, size reductions due to additional information about an argument are frequent. In fact, 58.9 % of methods have at least one precise or extant argument.

3. Problem and Proposed Solution

The inlining heuristic of Jikes RVM, briefly described in Section 2.2 above, has one conceptual flaw: It always reduces its size estimate for the callee when one of its arguments is of a precise type or pre-existent. It does so regardless of whether or not it is likely that some indirect benefit will manifest itself when the callee is inlined. For a callee which, e.g., simply stores its argument in a field or even ignores it altogether, knowledge about the precise type of said argument or about its pre-existence is worthless. In other words, the existing inlining heuristic of Jikes RVM makes no effort to predict whether the information about reference arguments can actually be used to good effect; it simply applies some blanket reductions to the callee's size (cf. Table 1).

Our proposed heuristic provides a partial remedy to this conceptual flaw: It makes a prediction based on information available in the dynamic call graph: Only if there exists an edge in the dynamic call graph which suggests that a method will be called on one of the callee's arguments and if the information known about that argument can be exploited, then inlining the callee should be encouraged, e.g., by applying a size reduction.

For purpose of illustration, consider the situation in Figure 1 again. Assuming the class hierarchy depicted in Figure 3, the dynamic call graph will contain an edge $\mathsf{B.n(C)} \rightarrow \mathsf{D.o()}$ if the former method calls the latter on its argument. However, such an edge may also be present for other reasons, e.g., if $\mathsf{B.n(C)}$ calls said method on a different instance. A heuristic that relies solely on call-graph profiles to decide the problem $\mathsf{A.m()} \overset{?}{\longleftarrow} \mathsf{B.n(D^\sharp <: C)}$ will occasionally operate under wrong assumptions. In the following, we are thus careful to use the phrase "an edge *suggests* that a method was called on the argument" when we codify the conditions under which information about the argument can be exploited.

Precise-induced Edges In the previous example, the precise type matched the edge's target class. However, the situation is not always as simple as this: When the heuristic decides the problem $\mathsf{A.m()} \overset{?}{\longleftarrow} \mathsf{B.n(G^\sharp <: C)}$ an edge $\mathsf{B.n(C)} \rightarrow \mathsf{G.o()}$ suggests an in-

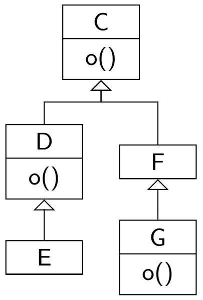

Figure 3. A class hierarchy with a virtual method o().

direct benefit whereas an edge $\mathsf{B.n(C)} \rightarrow \mathsf{C.o()}$ does not. This gives rise to the following definition.

Definition 3.1. *Given a method* $\mathsf{B.n(D^\sharp <: C)}$ *and a complete dynamic call graph, an edge* $\mathsf{B.n(C)} \rightarrow \mathsf{C'.o()}$ *with* $\mathsf{D <: C'}$ *is called* precise-induced *iff* $(\nexists\, \mathsf{E.o()})\ \mathsf{D <: E} \underset{\neq}{<:} \mathsf{C'}$.

In other words, if a method B.n(C) calls a method o() on its argument of precise type D, then the call graph will show an edge from B.n(C) to the first definition of o() found when traversing the class hierarchy from D upwards. This edge is called *precise-induced*.

Extant-induced Edges If the argument in question is extant rather than precise, the above definition has to be modified accordingly; the argument's dynamic type can now not only be any subtype of the argument's static type but also, in certain cases, one of its supertypes.

Definition 3.2. *Given a method* $\mathsf{B.n(C^\flat)}$ *and a complete dynamic call graph, an edge* $\mathsf{B.n(C)} \rightarrow \mathsf{D.o()}$ *with* $\mathsf{D <: C}$ *is called* extant-induced. *Furthermore, an edge* $\mathsf{B.n(C)} \rightarrow \mathsf{C'.o()}$ *with* $\mathsf{C <: C'}$ *is also called* extant-induced *if* $(\nexists\, \mathsf{C''.o()})\ \mathsf{C <: C''} \underset{\neq}{<:} \mathsf{C'}$.

Note that both of the above definitions refer to a *complete* dynamic call graph. Of course, in practice the dynamic call graph is often incomplete. However, the adaptive optimization system [3] of Jikes RVM only selects frequently executed methods for recompilation with the optimizing compiler. When the optimizing compiler has an inline decision to make, it is therefore probable that the callee has already been executed a number of times; thus, a call-graph edge probably already exists.

If deciding an inlining problem $\mathsf{A.m()} \overset{?}{\leftarrowtail} \mathsf{B.n(C)}$ our proposed heuristic therefore attributes a size reduction only if the dynamic call graph contains either a precise-induced or an extant-induced edge targeting a method on the argument in question. If no such edge exists, a positive decision of $\mathsf{A.m()} \leftarrowtail \mathsf{B.n(C)}$ is unlikely to result in further inlining.

However, even if further inlining takes place, this does not always mean that a size reduction is warranted for. In particular, non-virtual calls never require a guard to inline. Our heuristic therefore consults the dynamic call graph to ensure that the source of the (precise- or extant-induced) edge corresponds to either an invokevirtual or invokeinterface instruction before it attributes the corresponding size reduction (cf. Table 1). If the call was made using a invokestatic or invokespecial instruction instead, inlining it without a guard is already possible without additional information; a size reduction is unwarranted for.[3]

[3] Interestingly, in its method-size estimate (cf. Section 2.2), Jikes RVM treats virtual and non-virtual calls as incurring the same cost.

4. Evaluation

We next answer several questions regarding our proposed heuristic: What is the per-decision quality of the proposed heuristic and how does it compare to the default heuristic? And what are the heuristics' effects on compile time, program time, and overall runtime?

Our evaluation uses the popular DaCapo benchmark suite (release 2006-10-MR2) [5], although the eclipse and chart benchmarks occasionally had to be excluded for technical reasons. Unless otherwise noted, we use rigorous replay compilation [14] and perform matched-pair comparisons with 10 compilation plans per benchmark. To account for the influence of processor architecture, all measurements were performed on two quite different machines:

- A 2004 single-core AMD Athlon 64 3200+ processor clocked at 2.2 GHz, with 64 KiB L1 data and instruction caches, 512 KiB L2 cache, and 1024 MiB RAM running a 32-bit version of GNU/Linux (Kernel 2.6.32).
- A modern 4-core/8-thread Intel Core i7-870 processor clocked at 2.93 GHz with 4×32 KiB L1 data and instruction caches, 4×256 KiB L2 cache, 8 MiB L3 cache, and 4096 MiB RAM running a 64-bit version of GNU/Linux (Kernel 2.6.32).

A production configuration of Jikes RVM (SVN revision 16061) was used, in which most VM code is compiled ahead-of-time at the highest optimization level (O2). When compiling the boot-image, which contains the VM's code, we resort to the default heuristic of Jikes RVM; only code compiled at runtime is affected by our proposed heuristic. The VM's heap size was fixed at 512 MiB to reduce non-determinism due to garbage collection.

4.1 Per-Decision Quality of Inlining Heuristics

The heuristic proposed in Section 3 uses the notions of precise- and extant-induced edges to predict whether a given method will call another method on one of its precise or extant arguments. If this is the case, it is likely that the latter method can be inlined as well. In the following, we will evaluate how accurate these predictions are in practice by comparing them with the actual inlining decisions made by Jikes RVM's (default) inline oracle.

Our evaluation records all inlining problems decided by the default heuristic of Jikes RVM. Since our proposed heuristic never causes more inlining than the default, basing evaluation of the former on inlining problems encountered by the latter is reasonable.

Of all problems $\mathsf{A.m()} \overset{?}{\leftarrowtail} \mathsf{B.n(D^\sharp <: C)}$ and $\mathsf{A.m()} \overset{?}{\leftarrowtail} \mathsf{B.n(C^\flat)}$ decided during the benchmark runs, we exclude those where the default heuristic decides not to inline: $\mathsf{A.m()} \not\leftarrowtail \mathsf{B.n(C)}$. This is again reasonable, as further inlining becomes obviously impossible when there is no inlining in the first place. But if the default heuristic does decide to inline, there are four cases to consider:

True positive (t_+) The heuristic under evaluation correctly predicts further inlining suggestive of indirect benefits (cf. Section 3): $(\exists\, \mathsf{C'.o()})\ \mathsf{A.m()} \leftarrowtail \mathsf{B.n(C)} \leftarrowtail \mathsf{C'.o()}$, with a precise- or extant-induced edge $\mathsf{B.n(C)} \rightarrow \mathsf{C'.o()}$.

False positive (f_+) The evaluated heuristic incorrectly predicts such further inlining.

True negative (t_-) The evaluated heuristic correctly predicts that no such further inlining occurs.

False negative (f_-) The evaluated heuristic incorrectly predicts that no such further inlining occurs.

By classifying each prediction made by the heuristic under evaluation, we can now assess their quality using four standard metrics for information-retrieval systems shown in Figure 4: accuracy, F1-measure, precision and recall.

$$\text{Accuracy} = \frac{t_+ + t_-}{t_+ + f_+ + t_- + f_-} \qquad \text{Precision} = \frac{t_+}{t_+ + f_+}$$

$$\text{F1-Measure} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \qquad \text{Recall} = \frac{t_+}{t_+ + f_-}$$

Figure 4. Standard metrics for information-retrieval systems.

The accuracy is the ratio of correct predictions to the number of all predictions. The so-called F1-measure is the harmonic mean of precision and recall, where precision measures how often further inlining actually occurs when predicted by the heuristic and recall measures how often further inlining was predicted correctly by the heuristic when it actually occurs. The higher the precision, the less likely it is for the compiler to inline a callee just because of an unwarranted size reduction. The higher the recall, the more likely it is that the compiler exploits all opportunities for further inlining. For the proposed heuristic, low recall is caused by an incomplete dynamic call graph.

Using these metrics we can compare the proposed heuristic with the heuristic employed by the default inline oracle; the latter optimistically "predicts" further inlining for all precise and extant arguments. We also compare our proposed heuristic to a hypothetical random heuristic, which predicts further inlining and consequently awards size reductions with probability μ, which is the overall probability with which further inlining along a precise- or extant-induced edge actually occurs during benchmark execution.

Figure 5 summarizes the results for the three heuristics considered. The accuracy of the proposed heuristic is good, ranging from 89.94 % (jython) to 97.47 % (pmd). In fact, it is always better than the random heuristic. Both the proposed and the random heuristics outperform Jikes RVM's inlining heuristic by a large margin; the default heuristic's accuracy is at most 19.93 %.

The high accuracy of the proposed heuristic is mainly due to its high precision; whenever it predicts further inlining, such inlining is very likely to actually occur. Both the default and the random heuristic have equally low precision; the default heuristic always predicting further inlining is as likely to be correct as simply predicting this fact with probability μ. The default heuristic's low precision is also the reason for its low accuracy. In fact, its precision and accuracy are exactly the same, as the heuristic never makes a negative prediction—be it false or true negative.

In contrast to the proposed and random heuristics, however, the default heuristic has perfect recall. As it always awards a size reduction for precise- or extant arguments, it never misses an opportunity to perform further inlining. But while recall of the proposed heuristic is less than perfect, it is still much better than that of the random heuristic, which is again μ.

To summarize, the quality of the proposed heuristic is high: Both its accuracy and F1-measure are superior to the default and random heuristic.

4.2 Performance Measurements

As we have shown in the previous section, our proposed heuristic makes high-quality predictions. However, this does not necessarily mean that it improves program time. The slightly lower recall in particular may mean that our proposed heuristic misses important inlining opportunities which the default heuristic would have exploited. In the following, we thus present a series of performance measurements comparing both heuristics: the default heuristic of Jikes RVM and our proposal.

The implementation of our proposed heuristic is deliberately minimal. The presence or absence of precise- and extant-induced edges only determines whether the respective size reduction is awarded; all other reductions are awarded as before (cf. Table 1).

Performance with Replay Compilation The first set of measurements has been conducted using replay compilation [14], a methodology to control the non-determinism inherent in modern VMs that perform adaptive optimizations [3]. First, optimization decisions made during an invocation of the benchmark are recorded in a so-called compilation plan.[4] Then, these decisions are replayed according to the plan in a second invocation of the benchmark. This invocation's first iteration is called the mixed run, while all its other iterations are called stable runs. As we are primarily interested in the trade-off between program time and compile time, our performance evaluation considers the mixed runs only.

Figure 6 shows the results for mixed runs of all benchmark's except eclipse, which is incompatible with replay-compilation, on both the Athlon and Core i7 architectures.[5] Each scatter plot depicts the runtime of 20 runs conducted with 10 different compilation plans using the two heuristics of interest: the default heuristic of Jikes RVM and our proposed heuristic. In this matched-pair comparison [14] every compilation plan is therefore used twice, once by each heuristic. This ensures that the differences are indeed due to the heuristics used rather than due to different compilation advice. The fact that several benchmarks, e.g., antlr on the Athlon, exhibit outliers serves to illustrate how important it is to consider a large number of compilation plans in one's evaluation.

The scatter plots of Figure 6 show compilation time on the x-axis and program time, i.e., time spend outside of the just-in-time compiler, on the y-axis. The diagonals thus represent overall runtime; two runs placed on the same diagonal took the same time to complete, even if the time spend within and without the compiler differed. The closer a run is to the plot's lower left corner, the better its performance. In all plots, the distance of two diagonal lines represents 2.5 % of the average runtime of the respective benchmark when using the default heuristic. A higher line density thus indicates larger relative differences between individual runs.

As can be seen in Figure 6, using the proposed heuristic generally decreases compile time. This effect can best be observed for the bloat, hsqldb, and pmd benchmarks on the Athlon and for the bloat and luindex benchmarks on the Core i7. In a few cases, alas, this decrease in compile time is cancelled out by an increase in program time, an effect that can most clearly be observed for the pmd benchmark on the Athlon; the overall runtime stays almost constant. In other cases like bloat, luindex, or hsqldb, however, inlining less but more purposeful does not have a detrimental effect on pure program time; the decrease in compile time hence results in a decrease in runtime as well.

Figure 7 summarizes the results of Figure 6 for both the Athlon and Core i7 architectures by showing the per-benchmark speed-up. This per-benchmark speed-up is the geometric mean of the 10 individual per-plan speed-ups. As can be seen, for no benchmark besides jython does the proposed heuristic significantly degrade performance. In fact, for six benchmarks (bloat, chart, fop, hsqldb, luindex, and xalan) we observe a significant speed-up of up to 8.4 %. Also, the slowdown in the case of jython has to be taken with a grain of salt: As this benchmark uses a custom classloader which Jikes RVM does not handle properly during replay compilation, about two-thirds of the edges in the dynamic call-graph profile can never be matched as precise- or extant-induced edges; this severely disadvantages the proposed heuristic.

Performance without Replay Compilation The proposed heuristic relies on the dynamic call-graph profile to make its predictions. Thus, the closer the profile matches reality, the better predictions

[4] In Jikes RVM, each compilation plan consists of compiler advice, an intra-procedural edge profile, and an inter-procedural dynamic call graph profile.

[5] For technical reasons, results for chart are only available on the 32-bit version of GNU/Linux, hence on the Athlon architecture.

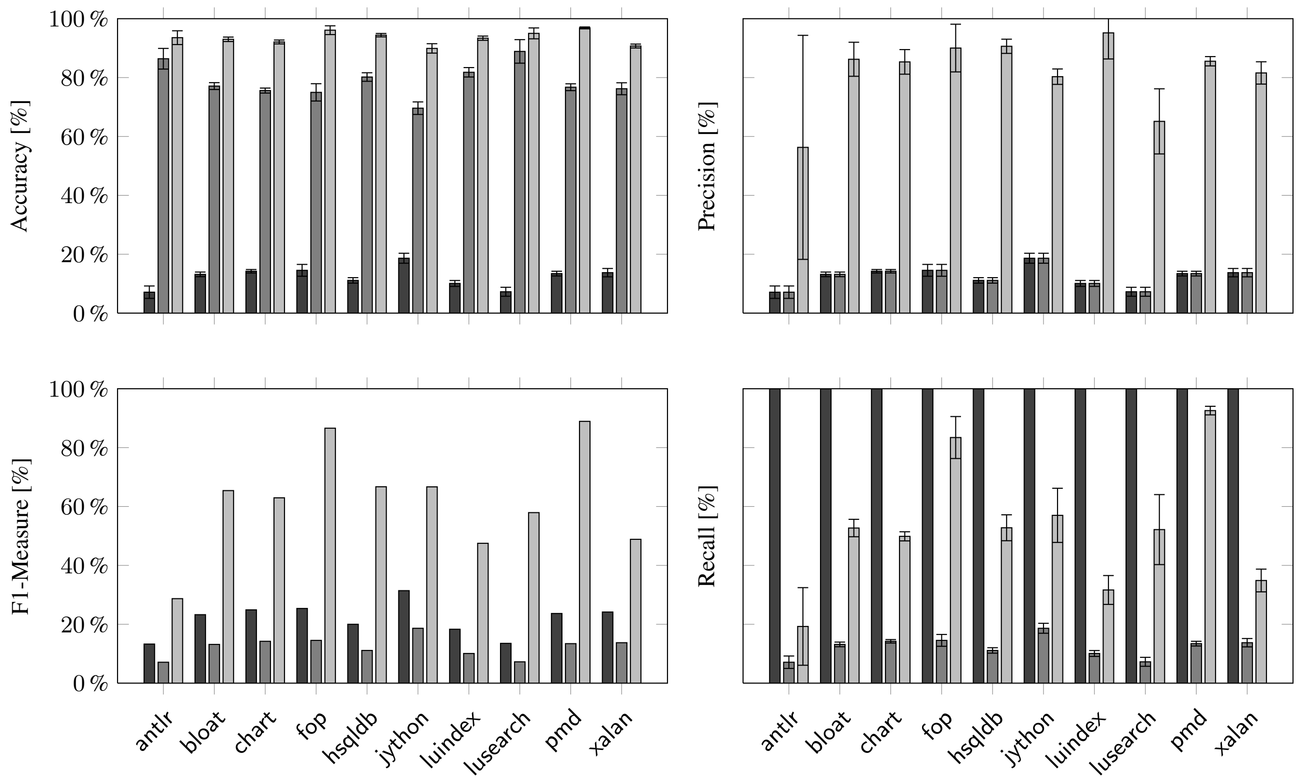

Figure 5. The per-decision quality of three inlining heuristics: the default heuristic of Jikes RVM (▮ ▮), the random heuristic (▮ ▮), and the proposed heuristic (▯ ▯).

the heuristic can make. Now, one effect of replay compilation is that the entire profile is available from the very beginning of the benchmark run. Compared to a non-replay run, during which the profile is build up over time, having the entire profile available immediately may overestimate the heuristic's effectiveness in a real-world, non-replay scenario. We have therefore conducted all our performance measurements also without replay compilation. Figures 8a and 8b show the results, whereby the former mirrors Figure 6 and covers those benchmarks for which replay compilation is supported and the latter covers the eclipse benchmark. Here, each scatter plot depicts 30 runs instead of the 20 runs per benchmark of Figure 6. This increase in the number of runs per benchmark and heuristic, from 10 to 15, compensates the higher degree of non-determinism inherent in measurements made without replay compilation [14].

As was the case when using replay compilation, the proposed heuristic significantly reduces compile time. Figure 9 shows the speed-up attainable in a non-replay setting. On the one hand, more time is spend compiling in such a setting, as methods may be recompiled several times. On the other hand, the dynamic call-graph profile is less complete than in a replay-setting, which reduces the proposed heuristic's effectiveness.

Note that benchmarks run on the Athlon architecture account for most of the speed-ups observed in this setting. This can be explain by the fact that on this architecture the ratio of compile time to program time is higher; Jikes RVM spends a larger portion of its time compiling. As the proposed heuristic works by reducing (unnecessary) inlining and hence compile time, the overall speed-up is higher on the Athlon than on the Core i7. Whether this bias can be avoided by better tuning the just-in-time compiler towards a particular architecture [18] and how this in turn affects the proposed inlining heuristic is beyond the scope of this paper.

5. Related Work

The cost-benefit inherent in inlining has been subject to much research, of which we can give only a brief overview here.

Grove et al. [15] investigated how offline profiles can be used to predict the receiver class at a given call-site; they then exploited this information by implementing profile-guided inlining. The authors also introduced the k-CCP (Call Chain Profile) model and found that inlining becomes the more effective the more calling context is available at a call-site. This observation is also at the core of this work. However, where the implementation of Grove et al. merely reaps the (indirect) benefits of further inlining, our work actively seeks such situations.

Hazelwood and Grove [16] subsequently implemented a context-sensitive inlining heuristic using online profiles. Their heuristic, like ours implemented in Jikes RVM, significantly reduced both compile time and code size at the expense of a minimal performance degradation. Compared to our work, their approach is again reactive rather than proactive.

Bradel and Abdelrahman [6] used offline profiles in the form of traces to direct inlining decisions. As the collected traces may span several methods, inline decisions can also account for the benefits of further inlining. While this approach leads to improved code quality, it substantially increases both compile time and code size. Also, unlike re-using the dynamic call graph to direct inlining decisions, collecting traces online would incur extra overhead.

Arnold et al. [4] modelled the cost-benefit trade-off inherit to inlining as a knapsack problem. Using this common framework, they compared three inlining heuristics based on the static call graph and the dynamic call graph with either node or edge frequencies. Using offline profiles, the authors approximated the globally opti-

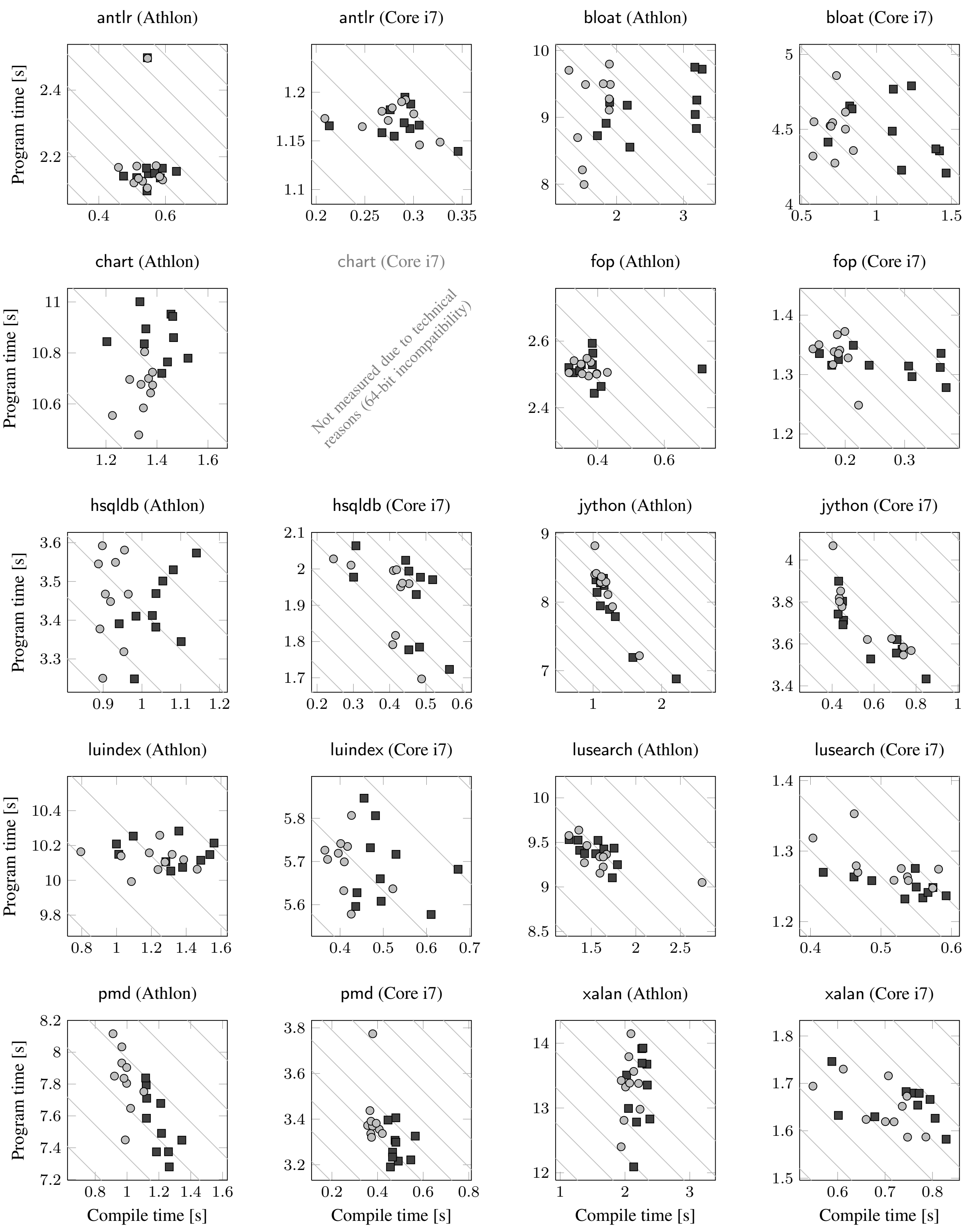

Figure 6. The effect of the default (■) and proposed heuristics (◎) on compile and program time. Each mark represents a benchmark invocation with a single compilation plan (10 plans overall). The diagonal lines represent total runtime.

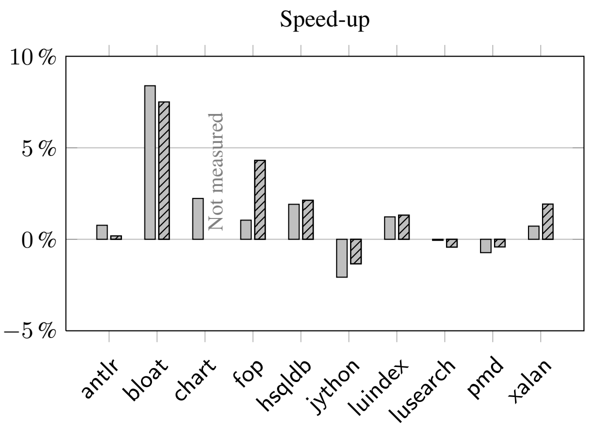

Figure 7. Speed-up achieved when using the proposed heuristic rather than the default heuristic of Jikes RVM on the Athlon (▯ ▯) and Core i7 (▨ ▨), respectively (Geometric mean of matched-pair speed-up for 10 compilation plans).

mal solutions to their respective inlining problems and found that a heuristic based on the dynamic call graph annotated with edge frequencies works best. Our approach, which also relies on the dynamic call graph, differs from the heuristics presented by Arnold et al. by taking indirect benefits into account; their heuristics only model the trade-off between increasing code size and decreasing the number of method calls.

Steiner et al. [20] compared a knapsack-based heuristic against two aggressive heuristics which inline in a depth-first or breadth-first fashion, respectively, until they reach an upper limit to either inlining depth and code expansion. In the comparison, the authors found that only the heuristic that performs a cost-benefit analysis, namely the knapsack-based heuristic, consistently improved performance. The authors remark, however, that "the hardest problem [...] is calculating good estimates of the benefits and costs of inlining [...]," a problem partly addressed by our work.

Cavazos and O'Boyle [7] used genetic algorithms to automatically tune Jikes RVM's inlining heuristics. In a similar fashion, Hoste et al. [18] used machine-learning techniques to automatically tune a just-in-time compiler, including its inlining heuristic. Both works report significant reductions in overall runtime. While highly effective, automated tuning of heuristics relies on the availability of useful features to the machine-learning algorithm. If a feature, e.g., the presence of call-graph edges suggestive of further inlining, is not exposed to the algorithm, it simply cannot be taken into account. Our work is thus complementary to such approaches.

Judging *a-priori* whether inlining a method offers any indirect benefits is difficult. Dean and Chambers [9] therefore proposed inlining trials, a technique in which the compiler experimentally inlines a method and keeps a record of how much information propagated from caller to callee was actually used for further optimizations. If another decision has to be made to inline such a method, these records are consulted to judge the indirect benefits possible in the current context. Using this technique, the authors were able to significantly reduce compile time with only minor increases in program time.

Suganuma et al. [21] present a similar approach to judge the benefits of specialization in a Java virtual machine: impact analysis. Here, at higher optimization levels the just-in-time compiler performs a special, static dataflow analysis. This allows for specialization decisions which take the additional information about the types and values of arguments and certain global variables into account. In contrast to the impact analysis of Suganuma et al., which relies on prior compilation of the method in question by the optimizing compiler, our inlining heuristic can already predict benefits when the optimizing compiler compiles a method for the first time. However, pushing a light-weight static analysis into Jikes RVM's baseline compiler might further improve the accuracy of our call-graph-based heuristic.

Dean et al. [10] developed an algorithm to judge the benefits of specialization in the Vortex compiler for Cecil. Like our heuristic, their algorithm bases its judgement on methods called on so-called pass-through arguments. Also, similar to our work, the judgement is based on the program's dynamic call-graph, which Dean et al. gather in a separate profiling run. Together with dataflow information about which arguments are indeed passed through, the Vortex compiler decides whether to specialize with respect to a given argument or not. The main difference between their work and ours is that between explicit specialization and the implicit specialization offered by inlining.

Further inlining is not the only indirect benefit one can reap when inlining a method; thus, the work by Shankar et. al [19] is only one example of a wider range of research. In it, the authors developed Jolt, a lightweight dynamic churn optimizer, which carefully guides inlining such that escape analysis becomes more effective and stack allocation of short-lived objects becomes possible.

6. Conclusions

In this paper, we have presented a heuristic which is able to better exploit inlining's indirect benefits by predicting when further inlining will be possible. To do so, our heuristic uses only information that is readily available in many modern virtual machines, namely an approximation of the dynamic call-graph; no static analysis or speculative compilation is required. This heuristic is highly accurate and can improve performance both in replay and non-replay settings by reducing compile time and code size. At the same time, its effect on code quality and hence program time is negligible.

This means, however, that the proposed inlining heuristic *ipso facto* does not improve code quality. This is because it only discourages inlining in cases where no indirect benefits due to further inlining are to be expected; it does not encourage inlining above and beyond what the original heuristic proposes. But being more conservative about one's inlining decisions may reduce register pressure and the number of instruction-cache misses. A detailed study of these positive effects, however, is subject to future work.

The proposed heuristic operates under the assumption that guardless inlining is the only indirect benefit caused by precise or extant arguments. If no edge in the dynamic call graph suggests that guardless inlining becomes possible, no size reduction is awarded. The elimination of dynamic type checks (`checkcast`, `instanceof`), however, is another indirect benefits that may manifest itself if additional information about arguments becomes available. Modelling this fact more accurately would require splitting the size reduction into two parts: a static part which is always awarded and which accounts for any type checks that might be eliminated and a dynamic part which is awarded based on the heuristic proposed in this paper.

Another way to further increase the heuristic's accuracy requires an enhanced dynamic call-graph profile: Whenever a method call is recorded, the enhanced profiler would also record which argument of the caller (if any) receives the call in question. This enhancement would do away with the need to approximate the possibility of further inlining through the notion of precise- and extant-induced edges,[6] albeit at the expense of some profiling overhead.

[6] Some Java virtual machines, most notably Oracle's HotSpot VM, do not build a dynamic call graph; thus, they would require such an enhancement.

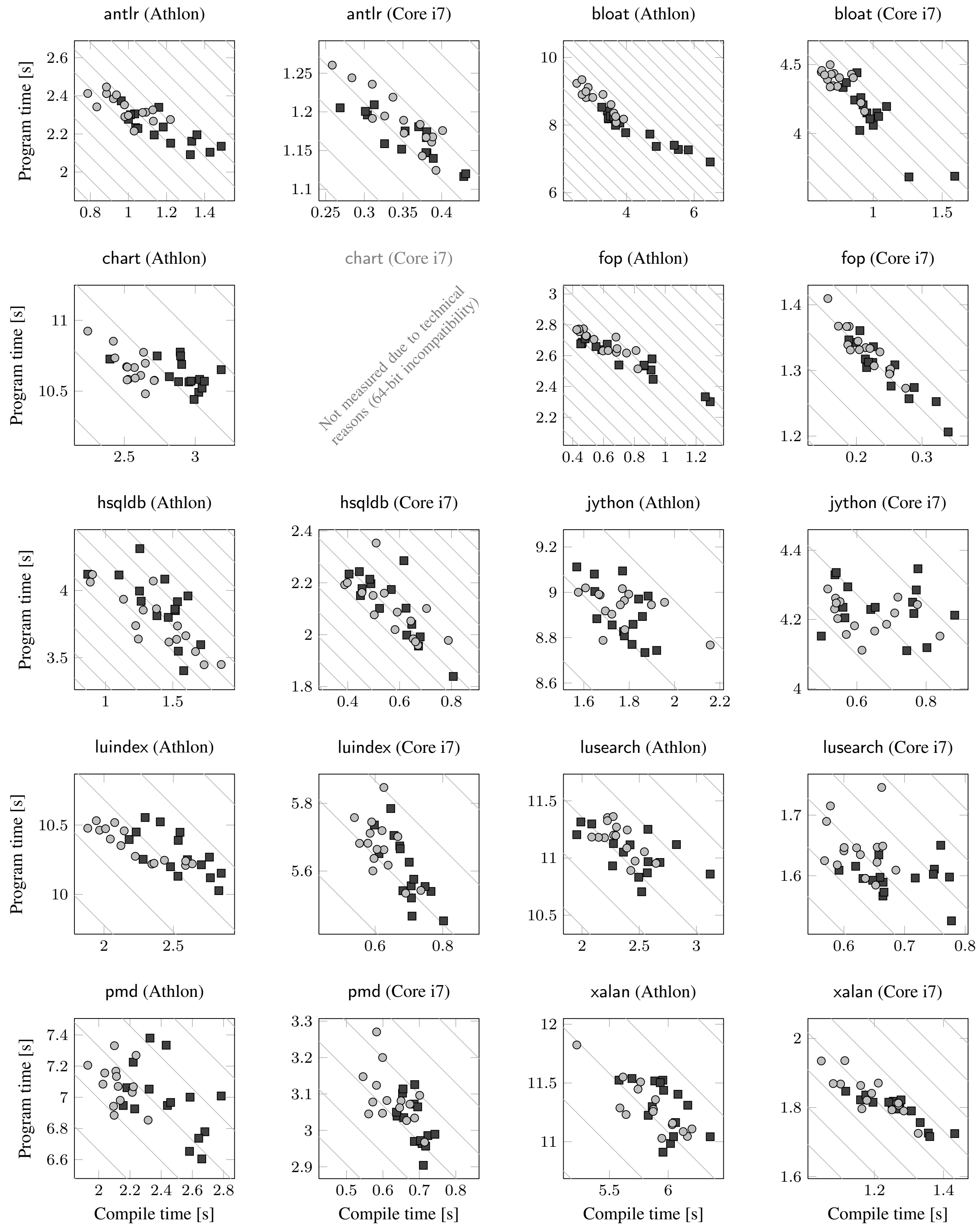

Figure 8a. The effect of the default (■) and proposed heuristics (●) on compile and program time. Each mark represents a benchmark invocation (15 invocations overall). The diagonal lines represent total runtime.

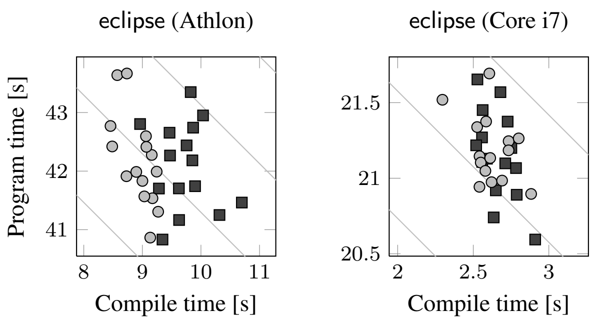

Figure 8b. The effect of the default (■) and proposed heuristics (◉) on compile and program time. Each mark represents a benchmark invocation (15 invocations overall). The diagonal lines represent total runtime.

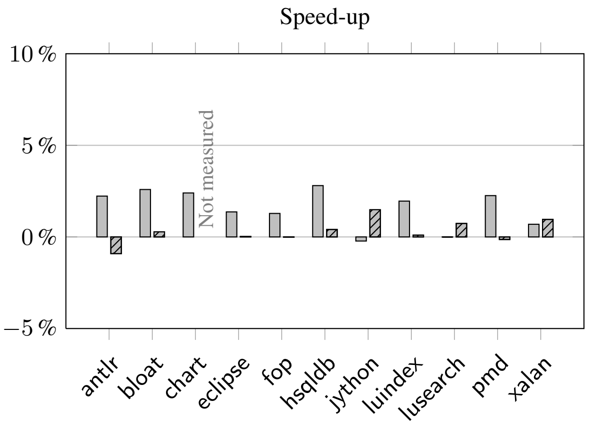

Figure 9. Speed-up achieved when using the proposed heuristic rather than the default heuristic of Jikes RVM on the Athlon (▯ ▯) and Core i7 (▨ ▨), respectively (Geometric mean of 15 invocations).

Acknowledgments

We would like to thank all anonymous reviewers for their extensive and encouraging feedback. Jan Sinschek provided helpful comments on earlier drafts of this paper.

This work was supported by the Center for Advanced Security Research Darmstadt (www.cased.de).

References

[1] B. Alpern, D. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, and J. J. Barton. Implementing Jalapeño in Java. In *Proceedings of the 14th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1999.

[2] M. Arnold and B. G. Ryder. Thin guards: A simple and effective technique for reducing the penalty of dynamic class loading. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2002.

[3] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the 15th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2000.

[4] M. Arnold, S. Fink, V. Sarkar, and P. F. Sweeney. A comparative study of static and profile-based heuristics for inlining. In *Proceedings of the Workshop on Dynamic and Adaptive Compilation and Optimization (DYNAMO)*, 2000.

[5] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2006.

[6] B. J. Bradel and T. S. Abdelrahman. The use of traces for inlining in Java programs. In R. Eigenmann, Z. Li, and S. Midkiff, editors, *Languages and Compilers for High Performance Computing*, volume 3602 of *Lecture Notes in Computer Science*, pages 922–922. Springer Berlin / Heidelberg, 2005.

[7] J. Cavazos and M. F. P. O'Boyle. Automatic tuning of inlining heuristics. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, 2005.

[8] M. Cierniak, G.-Y. Lueh, and J. M. Stichnoth. Practicing JUDO: Java under dynamic optimizations. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2000.

[9] J. Dean and C. Chambers. Towards better inlining decisions using inlining trials. In *Proceedings of the Conference on LISP and Functional Programming (LFP)*, 1994.

[10] J. Dean, C. Chambers, and D. Grove. Selective specialization for object-oriented languages. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 1995.

[11] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1995.

[12] D. Detlefs and O. Agesen. Inlining of virtual methods. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1999.

[13] S. J. Fink and F. Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2003.

[14] A. Georges, L. Eeckhout, and D. Buytaert. Java performance evaluation through rigorous replay compilation. In *Proceedings of the 23rd Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2008.

[15] D. Grove, J. Dean, C. Garrett, and C. Chambers. Profile-guided receiver class prediction. In *Proceedings of the 10th Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1995.

[16] K. Hazelwood and D. Grove. Adaptive online context-sensitive inlining. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2003.

[17] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 1994.

[18] K. Hoste, A. Georges, and L. Eeckhout. Automated just-in-time compiler tuning. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2010.

[19] A. Shankar, M. Arnold, and R. Bodik. Jolt: lightweight dynamic analysis and removal of object churn. In *Proceedings of the 23rd Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2008.

[20] E. Steiner, A. Krall, and C. Thalinger. Adaptive inlining and on-stack replacement in the CACAO virtual machine. In *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java (PPPJ)*, 2007.

[21] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a Java just-in-time compiler. In *Proceedings of the 16th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2001.