Adaptive Online Context-Sensitive Inlining

Kim Hazelwood
Harvard University
Cambridge, MA
hazelwood@eecs.harvard.edu

David Grove
IBM T.J. Watson Research Center
Yorktown Heights, NY
groved@us.ibm.com

Abstract

As current trends in software development move toward more complex object-oriented programming, inlining has become a vital optimization that provides substantial performance improvements to C++ and Java programs. Yet, the aggressiveness of the inlining algorithm must be carefully monitored to effectively balance performance and code size. The state-of-the-art is to use profile information (associated with call edges) to guide inlining decisions. In the presence of virtual method calls, profile information for one call edge may not be sufficient for making effectual inlining decisions. Therefore, we explore the use of profiling data with additional levels of context sensitivity. In addition to exploring fixed levels of context sensitivity, we explore several adaptive schemes that attempt to find the ideal degree of context sensitivity for each call site. Our techniques are evaluated on the basis of runtime performance, code size and dynamic compilation time. On average, we found that with minimal impact on performance (+/-1%) context sensitivity can enable 10% reductions in compiled code space and compile time. Performance on individual programs varied from -4.2% to 5.3% while reductions in compile time and code space of up to 33.0% and 56.7% respectively were obtained.

1. Introduction

Procedure inlining is perhaps the most important optimization for object-oriented programming languages such as Java$^{\text{TM}}$. The object-oriented programming style encourages decomposing a program into a large number of relatively small methods. Without aggressive inlining, this style can severely impact application performance due to both the direct costs of frequent procedure calls and the indirect costs of missed optimization opportunities caused by restricting the scope of optimization to such small program units.

Although procedure inlining can be very effective at reducing both the direct and indirect performance impact of an object-oriented programming style, it must be applied selectively. Overly aggressive inlining can greatly increase compile time and compiled code space, possibly even degrading application performance. This is especially true in the context of Java virtual machines, since compilation is typically performed at runtime by a Just-In-Time (JIT) compiler. In this dynamic context, the JIT must carefully balance the potential benefits of inlining with the compile time and code space costs.

One common technique to improve the quality of inlining decisions is to utilize profile information to identify frequently executed call edges [8, 15, 13]. Focusing inlining effort on the most frequently executed call sites increases the odds of maximizing benefit and minimizing cost. An additional benefit of profile-directed inlining is that it can enable guarded inlining of the dominant target(s) of a virtual dispatch when the distribution of receiver classes is skewed [6, 15, 13].

Most Java virtual machines rely on context-insensitive profile information to guide inlining decisions [3, 18, 23]. Previous work in offline profile-directed inlining has demonstrated that context-sensitivity can increase the benefits and reduce the costs of profile directed inlining [13]. In this paper, we study the effectiveness of adaptive, online, context-sensitive profile information to guide inlining decisions in the Jikes$^{\text{TM}}$ Research Virtual Machine (Jikes RVM). The main contributions of our work are:

- All profiling, decision making, and inlining are performed online. In an online system, decisions must be based on a limited history (program execution so far) and the profiling mechanisms must be highly efficient.
- We describe and evaluate several schemes for adaptive, context-sensitive profiling.
- We have implemented and empirically evaluated the described system in Jikes RVM version 2.1.1. On average, we found that with minimal impact on performance (+/-1%) context sensitivity can enable 10% reductions in compiled code space and compile time over Jikes RVM's existing implementation of context-insensitive profile-directed inlining. We plan to make our implementation available (open source) in a future release of Jikes RVM.

The remainder of the paper is organized as follows. Section 2 contrasts context-sensitive and context-insensitive profile information and provides a motivating example for our work. Section 3 provides the implementation details by describing the internals of the Jikes RVM adaptive optimization system and highlighting the changes necessary for implementing context-sensitive inlining. Section 4 describes a number of adaptive policies for deciding the level of context sensitivity to apply on a case-by-case basis. Experimental results are analyzed in Section 5. Finally, Section 6 describes the related work and Section 7 concludes.

2. Context Sensitivity Overview and Motivation

Systems that employ method inlining typically use profile information to guide their inlining heuristics. In fact, most Java Virtual Machines use profile information gathered at runtime. However, the profile information collected at runtime is at the granularity of a single call edge. This is typically referred to as *context-insensitive edge profiling*. Using this profiling scheme, inlining decisions are made based on the profile associated with the call edge between the callee method (the inlining candidate) and the caller method (the destination of the inlined code), with no additional information about the manner in which the call site was reached.

Previous work on offline optimization of object-oriented languages demonstrated that using *context-sensitive trace profiling* can enable significant improvements in the effectiveness of profile directed inlining [13]. The goal of our work is to achieve these benefits online in a virtual machine. The primary challenge of an online system is that all decisions must be made without knowing the future: only the profile information from the current execution of the program *so far* is available. Furthermore, both the profiling mechanism itself and the decision making process must be highly efficient since both occur during program execution.

Context sensitivity implies tailoring an analysis or optimization to specific times or locations during program execution. By taking into account *how the program arrived* at a particular point in the execution, the system can often make more informed optimization decisions. For example, by paying attention to certain actions leading up to a virtual method call, we may be able to predict the target method that will be invoked from the virtual call site. Figure 1 shows a small program where context sensitivity can improve the accuracy of the profile data. In this example, the `main` method sets up a simple hash table, inserts two elements, then calls `runTest`. `runTest` then makes two calls to the `HashMap.get` method, which calls the `hashCode` method on `key`. During the first call to `HashMap.get`, this call invokes `MyKey.hashCode`, and during the second call it invokes `Object.hashCode`.

Figure 2a shows a partial call graph of our sample program. A context-insensitive edge profiling system collects single call edges (i.e. method *A* calls method *B*) and associates profile information with those edges. Therefore, all inlining decisions that utilize the profile information will be based on a single edge of call history. In our HashMap example, any decision to inline the `hashCode` method into the `HashMap.get` method will be based on the profile information shown in Figure 2b. Since the weight of the two call edges indicates a 50/50 split, the the inlining system will either (a) inline both versions of the `hashCode` method at each call site, or (b) inline neither version. In this example, the profile information delivered by the context-insensitive profiling system is clearly misleading, resulting in a suboptimal inlining decision.

On the other hand, a context-sensitive profiling system would collect the profile shown in Figure 2c. From this figure, we can clearly see that all calls from `HashMap.get` to `hashCode` that originated from the first call site in `runTest` always evaluate to `MyKey.hashCode`. Therefore, instead of inlining both version of `hashCode` at both call sites, we will inline the correct version at each call site. By more precisely predicting the target of the virtual call, context sensitivity both reduces code space and compile time and increases application performance. The call to `equals` within `HashMap.get` also benefits from context sensitivity in exactly the same way. Although this particular program may seem artificial, it illustrates a common situation that arises in programs that make use of the standard Java collection classes.

Context-sensitivity can also improve inlining results for non-virtual calls. For example, a call site might be control dependent on the value of a parameter to its enclosing method: in some contexts the call is always executed, in others it is never executed. If the enclosing method is itself inlined into some of its callers, context-sensitive profile information could help avoid uselessly inlining the target of the call site in contexts where it will never be executed.

3. Implementation

The ideas described in this paper were implemented in the Jikes RVM adaptive optimization system [3]. Figure 3 shows the structure of the system, including our additions and modifications which are highlighted by dotted lines. This section describes the previous system structure and the changes that were necessary to implement context-sensitive inlining.

3.1. Inlining in the Optimizing Compiler

Previous work in Jikes RVM [2] developed a clean separation between the inlining mechanisms in the optimizing compiler and the inlining policy that determines where and how to apply inlining. The inlining policy is encapsulated in an Inlining Oracle abstraction that the compiler consults for each call site to determine which callees, if any, should be inlined. For our experiments, we created a new implementation of this policy module based on an existing one that supported context-insensitive profile-directed inlining [3]. We did not need to otherwise change the optimizing compiler.

```
class HashMapTest {
  static int counter;
  public static void main(String[] args) {
    Object k1 = new MyKey(22);
    Object k2 = new Object();
    HashMap map = new HashMap();
    map.put(k1, new Integer(1));
    map.put(k2, new Integer(2));
    runTest(k1, k2, map);
  }
  public static void runTest(Object k1, Object k2, HashMap map) {
    counter += ((Integer)map.get(k1)).intValue();
    counter += ((Integer)map.get(k2)).intValue();
  }
}

class MyKey {
  int key;
  MyKey(int k) { key = k;  }
  public int hashCode() { return key; }
  public boolean equals(Object other) {
    return other instanceof MyKey && ((MyKey)other).key == key;
  }
}

class HashMap {
  ...
  // simplified version of HashMap.get
  public Object get(Object key) {
    int index = (key.hashCode() & 0x7FFFFFFF) % elementData.length;
    HashMapEntry entry = elementData[index];
    while (entry != null) {
      if (entry.key == key || key.equals(entry.key)) return entry.value;
      entry = entry.next;
    }
    return null;
  }
}
```

Figure 1. An example of how context-sensitive profiling can improve the effectiveness of inlining for methods of Java's collection classes. Context-insensitive profile-directed inlining will either inline both `Object.hashCode` and `MyKey.hashCode` into `HashMap.get` at both call sites in `runTest` or will inline neither. Context-sensitive profile-directed inlining will inline `MyKey.hashCode` into `HashMap.get` at the first call site in `runTest` and `Object.hashCode` into `HashMap.get` at the second call site in `runTest`.

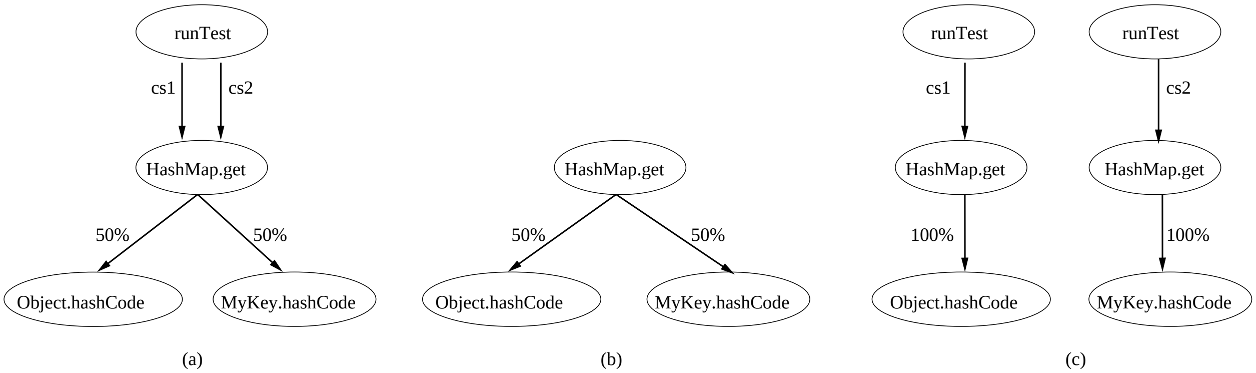


Figure 2. Call graph of HashMap example including profile weights. Part (a) shows the actual program call graph. Part (b) shows the profile information that context-insensitive profiling would collect. Part (c) shows the profile data that context-sensitive profiling would collect.

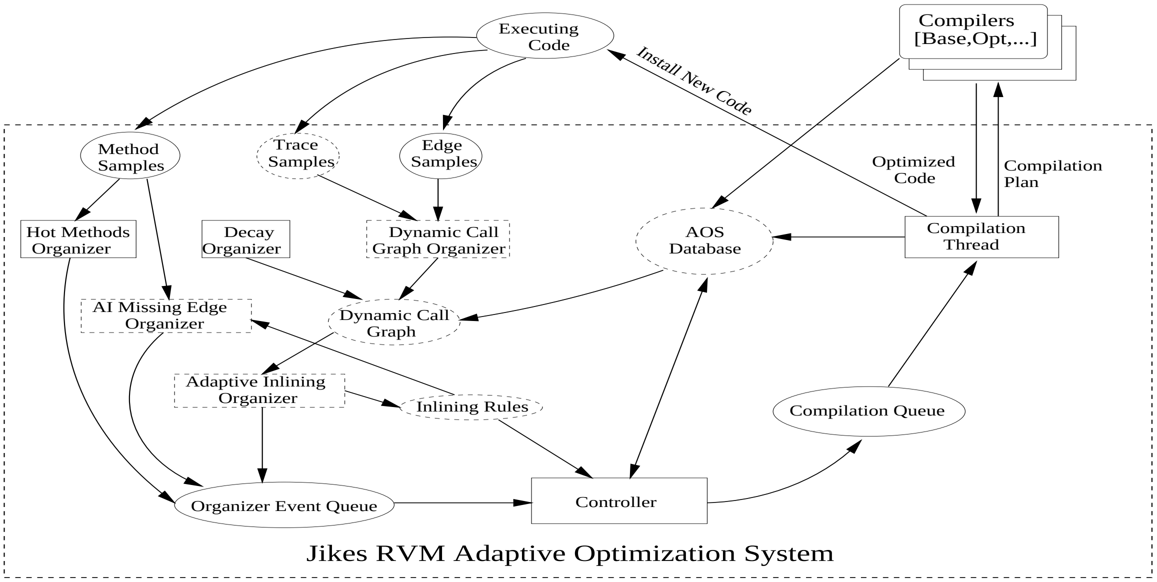


Figure 3. Implementation of feedback-directed inlining in Jikes RVM

Jikes RVM classifies methods into four categories based on an estimate of the size of the machine code that would be generated when the method is inlined.[1] Tiny methods, whose method body is estimated to be smaller than 2x the number of instructions required for a method call, are unconditionally inlined if they can be statically bound without a guard. Small methods, whose size ranges from 2-5x larger than a method call, that can be statically bound (possibly with a guard) are inlined subject to code space expansion and inlining depth heuristics. Medium-sized methods, whose size range from 5-25x that of a call, are only candidates for profile-directed inlining. Methods larger than 25x a call are never inlined.

Profile data is used to augment these static heuristics in three ways.

- To enable guarded inlining at call sites where the combination of class analysis [16, 7], class hierarchy analysis [9, 12, 11] and pre-existence [10], was unable to determine that a single target was possible at a virtual or interface invocation.
- To inline a medium-sized method.
- To inline small methods when the normal limits on code expansion and inlining depth have been exceeded.

3.2. Adaptive Inlining Subsystem Overview

The structure of the adaptive optimization system of Jikes RVM [3] is shown in Figure 3. All dynamically compiled methods are initially compiled by a non-optimizing baseline compiler. The role of the adaptive optimization system is to identify and selectively optimize program hot spots.

The *controller* is the decision-making component of the adaptive optimization system. It reads the information in the organizer event queue and uses an analytic model to decide what new compilation steps should occur. When a method is selected for recompilation, a compilation plan is created that includes an Inlining Oracle object that encapsulates the applicable inlining rules.

The controller's compilation decisions are inserted into the *compilation queue* to be read by the *compilation thread*. This thread executes the controller's compilation plan and installs the resulting new code into the executing program.

Profile information is gathered via timer-based sampling. Sampling occurs on each virtual processor at a rate of approximately 100 samples per second. Samples are taken by *listeners*, such as the *method* or *edge* listeners shown in the figure. Each time it takes a sample, the method listener records the currently executing method; this information is used to identify hot methods. Each time a sample is taken in a method prologue, the edge listener inspects the call stack and records a tuple of the form

$$\langle caller, callsite, callee \rangle \tag{1}$$

representing the call edge that led to the currently executing method. This information is used to identify hot call edges for inlining. The listeners insert this raw data into buffers to be read by the various *organizers*. Periodically the organizers process the buffers and convert the raw profile data into a form that can be easily understood and propagated through the rest of the system.

[1] The compiler adjusts this estimate to model the expected impact on inlined code size of the dataflow properties computed for the call's actual parameters. For example, if one of the parameters is a constant then the inlined size estimate is reduced to model the likely effects of constant folding.

Several organizers are fed by the two listeners. They include the *hot methods organizer*, the *dynamic call graph organizer*, and the *AI missing edge organizer*.[2] The hot methods organizer aggregates the samples taken by the method listener; this data drives the controller's recompilation decisions. The dynamic call graph organizer collates the raw data collected from the edge listener to create a profile of the call edge tuples of the program. This call graph information is then passed on to the *adaptive inlining organizer* where the edges are analyzed to generate a set of inlining rules (edges that should be inlined if possible). The AI missing edge organizer periodically examines the current set of hot optimized methods and inlining rules to determine if an opportunity to inline a hot edge has been missed because the edge became hot after the method was last compiled.

The *decay organizer* periodically decays the profile dynamic call graph to bias hot edge detection toward recently sampled call edges. Thus, the decay organizer attempts to ensure that the system can adapt to program phase shifts.

The *AOS database* is a central repository for recording and querying various compilation decisions and events. One use of this repository is by the inlining system to record refusals by the optimizing compiler to inline particular call edges. This information is used by the AI missing edge organizer to avoid recommending a method for recompilation due to a hot call edge that the optimizing compiler has already refused to inline.

3.3. Implementation Details

This section describes our modifications to the adaptive optimization system to support adaptive, context-sensitive inlining. Figure 3 shows that we added an additional listener to the system, namely the *trace listener*. This new listener was introduced because a single call is insufficient for context-sensitive inlining. While the edge listener collected a set of tuples (as shown in Equation 1), the data gathered by the trace listener is of varying size. The trace itself is a linear sequence of calling methods that lead to the current call site. The structure of the sample has the form

$$\langle caller_1, callsite_1, \cdots, caller_n, callsite_n, callee\rangle \quad (2)$$

where n can be either a fixed or adaptive value as described in Section 4.2–4.3. Each of these trace samples are inserted into a buffer until the desired number of samples has been reached, then the dynamic call graph organizer is notified.

The dynamic call graph organizer was extended to handle the call stack trace structures produced by the trace listener. Dealing with varying levels of profile information was a challenging problem, and several interesting issues arose. Specifically,

- How do we match profile information for variable-depth call traces? Should we group profile information for $A \Rightarrow C \Rightarrow D$[3] in with that for $F \Rightarrow A \Rightarrow C \Rightarrow D$?
- Some call traces may be deceiving because previously inlined call sites will be "missing" from the stack frame. How do we account for past inlining actions when sampling the call stack?

Partial Context Matches The previous system employed context-insensitive edge profiling, where edges were represented as tuples (Equation 1). Therefore, collecting profile information was a simple matter of maintaining a hash table for all tuples encountered during profiling, and updating counters associated with each tuple. As each new edge sample was collected, the system searched the hash table for an exact match of the tuple (inserting a new entry as necessary) then incremented the edge count for the particular tuple. Similarly, it was easy to determine whether profile data was applicable to a given call site during compilation: either the $\langle caller, callsite\rangle$ portion of the tuple matched exactly or it was not applicable.

Introducing variable-sized call traces into the profiling system significantly complicated the data structure and the search technique, introducing the notion of a *partial trace match*. A partial trace match occurs when profile information is collected on a call trace that happens to be a subset (or superset) of an existing call trace in the profile data. The question arises whether to allow partial matches and combine the two profiles, or to instead maintain separate traces in the profile data.

Our current solution is a hybrid approach. When profile data is collected and hot traces are identified and codified as inlining rules, we do not merge partial matches. However, when the inline oracle is determining which inlining rules are applicable to a candidate call site in a particular compilation context it allows partial matches. If the compilation context is $\langle caller_{ck}, callsite_{ck}, \cdots, caller_{c1}, callsite_{c1}\rangle$ then all inlining rules with context $\langle caller_{tj}, callsite_{tj}, \cdots, caller_{t1}, callsite_{t1}\rangle$ such that

$$\begin{aligned}&\forall i\ 1 \leq i \leq min(k, j),\\ &\qquad caller_{ci} = caller_{ti} \wedge callsite_{ci} = callsite_{ti} \quad (3)\end{aligned}$$

are applicable. To determine which methods should be inlined at the call site, the oracle first constructs sets of target methods of all applicable traces with identical contexts. All methods in the intersection of these sets are identified as inlining candidates. Intuitively, if a callee method has been frequently invoked from all traced contexts that are applicable to the context being compiled, then we predict that it is a good candidate for inlining, even if there is no hot trace that exactly matches the compilation context. Allowing partial matches is important because it is often the case that the profile data has more (often irrelevant) context than is available at the call site being compiled.

[2] AI is an abbreviation for Adaptive Inlining

[3] Throughout the paper, we use the symbol ⇒ to indicate a method call, therefore $A \Rightarrow B$ denotes "method A calls method B"

Optimized Stack Frames Because of inlining, a single optimized stack frame may actually represent an arbitrary number of source level stack frames. For example, profile information may be present for the call trace $A \Rightarrow B \Rightarrow C$, but due to an online inlining decision, *B* may have been inlined into *A*. At this point, a naive trace listener might record the trace sample $A \Rightarrow C$. This sample is misleading because it appears to be different from the profile data for $A \Rightarrow B \Rightarrow C$, when in actuality, the two should be combined.

Fortunately, Jikes RVM already supported mechanisms to recover the source level view of optimized stack frames (required to support class loading, security managers, and debugging). The trace listener was able to utilize these mechanisms to correctly sample optimized stack frames.

4. Context-Sensitive Profiling Policies

While it seems logical that in many cases, more than one edge of profile information may be helpful in making inlining decisions, we can also envision cases where too much context sensitivity may degrade performance. Slowdown may either result directly from the overhead of collecting and analyzing context-sensitive profile data, or indirectly from diluting profile information such that it takes longer for the system to decide that a call is hot and should be considered for inlining. Our experimental results indicate that this second effect, profile dilution, is actually quite important. The AI organizer constructs inlining rules by examining all edges/traces that contribute more than a threshold percentage of the total weight of the profile data.[4] As additional levels of context sensitivity are added to the profile data, the profile weight of a single context-insensitive call edge can be distributed among an increasing number of traces. If the weight is distributed unevenly, then this is useful because context sensitivity is discriminating the important and unimportant instances of the call edge. However, if the weight is distributed more or less evenly, then it will take the system longer to arrive at exactly the same inlining decisions. For this reason, it is important to locate the particular call sites that will benefit from additional context sensitivity, then use it selectively for those call sites. The following sections describe various schemes for finding the best amount of context sensitivity. These policies are grouped into three categories: *ideal*, *fixed-level*, and *adaptive*.

4.1. Ideal Sensitivity

For each dynamic invocation of a method, there is a notion of the *ideal* amount of context sensitivity. This is the point when the context in which a method is called no longer *matters*. More specifically, it is the highest point in the dynamic call graph where actions are taken that affect the behavior of the current method or call site. Finding this point is in general undecidable.

One possible approach that might closely approximate this ideal would be to analyze each method and identify call sites that are data or control dependent on parameters to the method. These call sites would then be flagged as requiring additional context when sampled. As the listener sampled the stack, it would continue to trace the stack until it encountered a call site that was not flagged as requiring additional context.

4.2. Fixed-level Sensitivity

Just as in program analysis, the simplest context-sensitivity policy is to set the context-sensitivity level of all compiled methods to a fixed value [22]. We implemented this policy and studied the performance of the system as we varied the value from *2* to *n*. A context-sensitivity level of *2* means that profile information is collected for sequences of two call edges. Thus, instead of associating a profile weight with the call sequence $C \Rightarrow D$, we would collect information on the sequence $A \Rightarrow C \Rightarrow D$, and would distinguish that from $B \Rightarrow C \Rightarrow D$.

4.3. Adaptive Sensitivity

However, as discussed earlier, too much context sensitivity can be counterproductive. Furthermore, the desired amount of context sensitivity varies from call site to call site in the program. Thus, instead of using a fixed level of context sensitivity throughout the program, it is desirable to adaptively tailor the context-sensitivity policy for each call site on a case-by-case basis. This section describes five such adaptive policies. The first three are early-termination rules for a basic fixed-level policy. The intuition behind these policies is that they may enable us to use a fixed-level policy with a fairly large *n* value by ending the trace early (sampling fewer than *n* levels) when it is obvious that the additional levels will not be useful. The final two policies are hybrid policies with multiple early-termination rules.

Parameterless Methods One heuristic we explored focuses on *parameterless method calls*. The premise is that the context in which a caller method makes a call is inconsequential if there are no parameters to pass state from the caller to the callee. While there are certainly exceptions, such as global variables and the `this` parameter, this may be an inexpensive approximation to the ideal policy.

An obvious question is, *How often will this early-termination condition actually be triggered?* To answer this, we instrumented the trace listener to record the number of stack frames it traversed as it took each sample. For our benchmarks (described in Table 1), we found that 20% of sampled callee methods are immediately parameterless and would require no additional context sensitivity. We also discovered that most sampled traces contain a parameterless

[4] In our experiments, we used a threshold value of 1.5%

call within five levels of call stack (between 50% and 80%.) This shows us that this scheme will often limit the amount of context sensitivity to evaluate for our inlining decisions.

Class Methods In object-oriented languages, state may be implicitly passed from caller to callee via the instance fields (and type) of the `this` parameter. It may be that this state is more important than that passed by other parameters to the call. Therefore we investigated an early termination heuristic that ends the trace as soon as a class method call is encountered. We discovered that in 50–80% of the cases, we only traverse two edges in the call graph before encountering the first class method call.

Large Methods The final early termination condition we explored is quite different than the two previous ones. Rather than focusing on the flow of state from caller to callee, it instead focuses on a feature of the inlining system itself. As mentioned in Section 3.1, the inlining system never inlines *large* methods. Because of this, we may consider limiting our profiling to one level above a large method in the call chain, since a large method will never be inlined into a parent method. From our benchmarks, we discovered that roughly half of the time, we had to traverse four or more call edges before encountering our first large method.

Hybrid Early-Termination Policies These three early termination polices can obviously be combined to form hybrid policies. We implemented two of the possible combinations: Parameterless Class Methods and Parameterless Large Methods.

Adaptively Resolving Imprecisions Our final adaptive policy takes a very different approach. It starts by collecting context-insensitive profile information for all call sites. As the profile data is processed, the dynamic call graph organizer identifies polymorphic virtual call sites that do not have highly skewed distributions of callees. Without more precise profile information it will not be feasible to inline the targets of these call sites. Therefore, these call sites are flagged as requiring additional levels of context sensitivity to identify contexts in which they could become potential inlining candidates. This process continues until either the imprecisions in the profile data are resolved or the system determines that the call site is inherently too polymorphic. Plevyak's iterative algorithm for call graph construction used a similar approach to context sensitivity in an offline setting [19].

We have not yet implemented this policy, but think that it has promise. An open question is whether it is possible to do this iteration online without incurring significant overhead or delay before finally arriving at useful profile information.

5. Experimental Results

All experiments were performed using the Jikes Research Virtual Machine version 2.1.1 on a Pentium-3 based Red Hat Linux 7.2 workstation with 2.5 GB RAM. We used the FastAdaptiveSemispace configuration of Jikes RVM, indicating that the core virtual machine was compiled with all assertions disabled by the optimizing compiler, that initially all dynamically loaded methods are compiled by the non-optimizing baseline compiler and the adaptive optimization system is used to select a subset of the dynamically compiled methods for optimizing recompilation, and that the basic semispace copying collector was used. The benchmarks selected were the SPECjvm98[5] benchmark suite and the SPECjbb2000 benchmark.

In this section, we explore the performance of six of our context-sensitivity policies in terms of overall runtime performance, size of the dynamically optimized code, and runtime overhead of the adaptive optimization system. As inlining policies must carefully balance performance, code size and compile time, we investigate and compare these values.

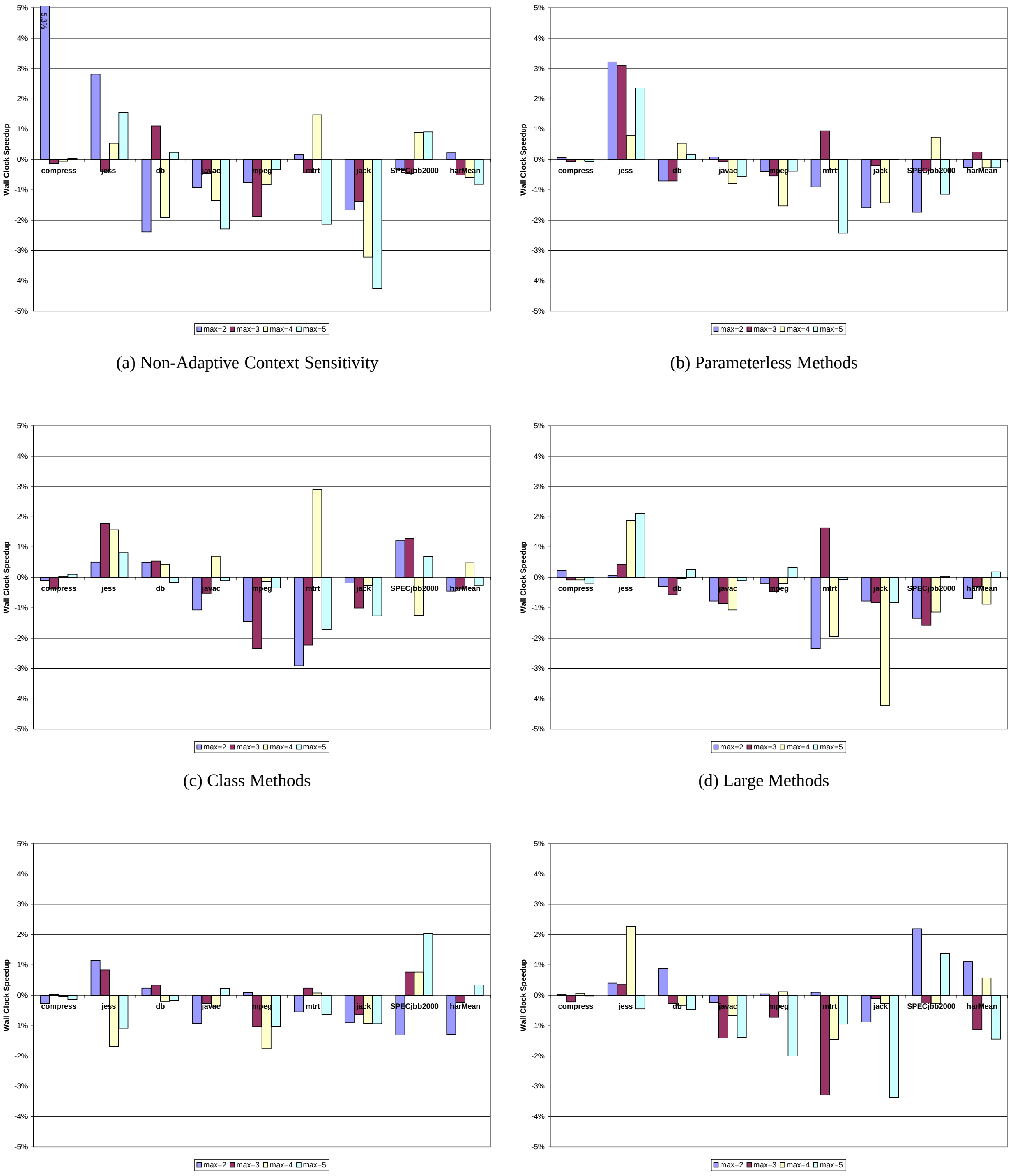
We explore the performance of the Fixed Sensitivity policy, while varying the sensitivity from *2* to *5* call edges. Then we compare this performance to three adaptive policies – Parameterless Methods, Class Methods, and Large Methods. For each of the adaptive (early-termination) policies, we again vary the maximum sensitivity from *2* to *5*. Finally we explore two hybrid schemes – Parameterless Class Methods and Parameterless Large Methods.

Figure 4 depicts the wall-clock performance speedup of each of the policies as a percentage improvement over the existing context-insensitive inlining policy implemented in Jikes RVM. Figure 5 illustrates the change in optimized code size resulting from the profiling policies we implemented. The most notable observation from these figures is nearly all benchmarks benefited from a reduction in code size, with minimal impact on runtime performance.

The performance impact of context sensitivity was smaller than one might expect, partially because of the benchmark suite. From previous experiments with the context-insensitive inlining implementation in Jikes RVM, we know that in most of the SPECjvm98 benchmarks, context-insensitive profile data is sufficiently precise that dynamically, no guarded inlining of a virtual call completely fails (none of the inlined methods matches and execution reaches the fallback virtual invocation.) Therefore, for these benchmarks we are only expecting context sensitivity to reduce code space and compile time, although it might enable a slight performance gain by reducing the number of inline guards executed before the correct inlined target is found.

One case where reducing inline guards has made an impact on performance is in `jess`. Because `jess` has a

[5] These benchmarks were developed by the Standard Performance Evaluation Corporation [24]. The performance numbers reported in this paper are the best run of 20 on each individual SPECjvm98 benchmark. These runs do *not* conform to the official SPEC run rules, so our results do not directly or indirectly represent a SPECjvm98 metric, and are not comparable with a SPECjvm98 metric. Due to the use of timer-based sampling, the adaptive optimization system is non-deterministic. Therefore a single performance execution is not necessarily indicative of overall performance trends.



(a) Non-Adaptive Context Sensitivity

(b) Parameterless Methods

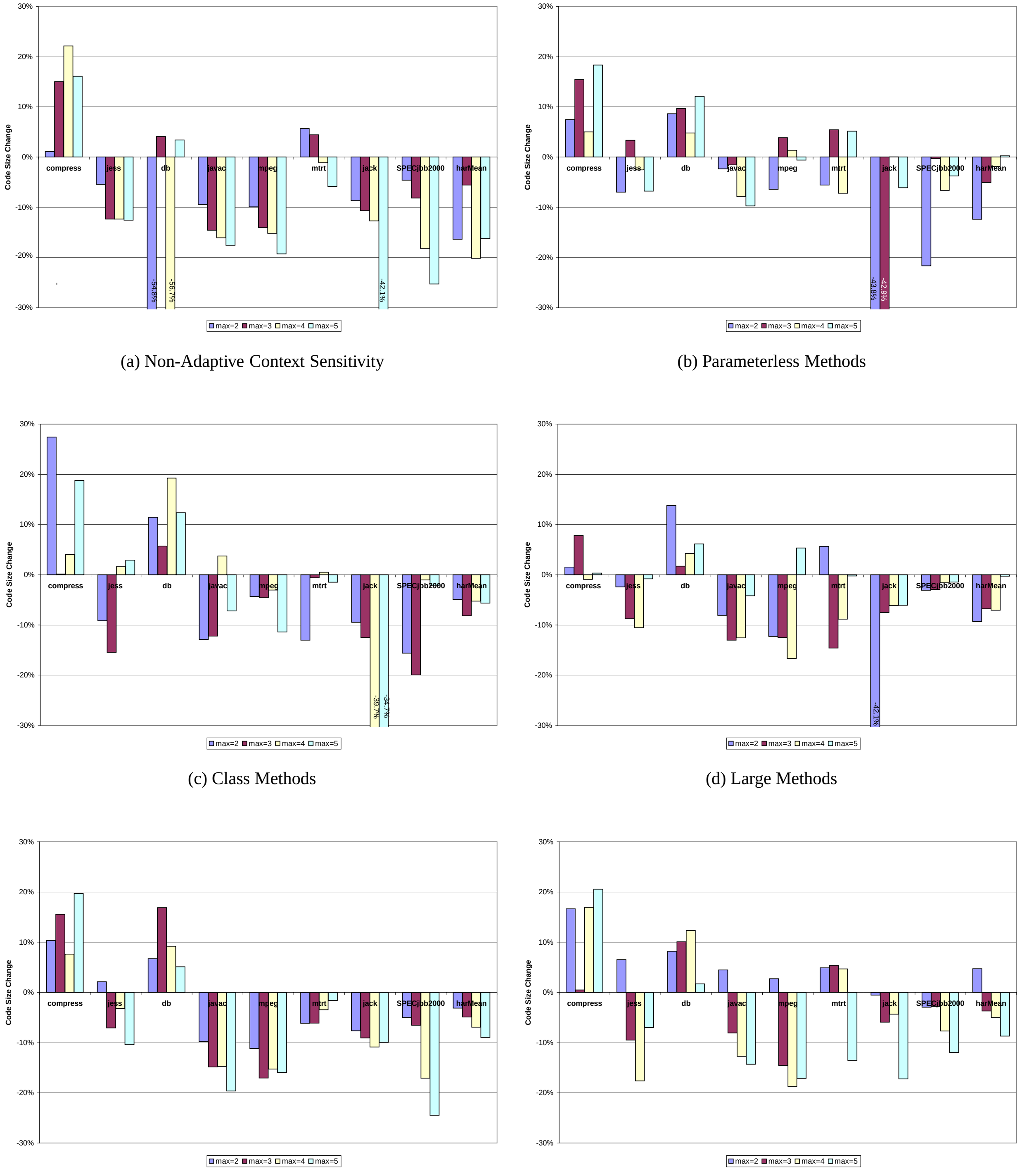
(c) Class Methods

(d) Large Methods

(e) Hybrid 1 - Parameterless Class Methods

(f) Hybrid 2 - Parameterless Large Methods

Figure 4. Wall-clock speedup of each of our implemented context-sensitive profiling policies. The y-axis represents speedup as a percentage improvement over context-insensitive inlining as previously implemented in Jikes RVM. The four bars for each policy represent the performance as the maximum context sensitivity is varied from *2* to *5*.



(a) Non-Adaptive Context Sensitivity

(b) Parameterless Methods

(c) Class Methods

(d) Large Methods

(e) Hybrid 1 - Parameterless Class Methods

(f) Hybrid 2 - Parameterless Large Methods

Figure 5. Code size changes resulting from each of our context-sensitive profiling policies. The y-axis represents the percent increase in bytes of optimized machine code over context-insensitive inlining as previously implemented in Jikes RVM. (Negative numbers are desirable.) The x-axis shows the maximum context sensitivity allowed for each policy.

Benchmark	Description	Classes	Methods	Bytecodes
compress	Lempel-Ziv compression algorithm	48	489	19,480
jess	Java expert shell system	176	1101	35,316
db	Memory-resident database exercises	41	510	20,495
javac	JDK 1.0.2 Java compiler	176	1496	56,282
mpegaudio	Decompression of audio files	85	712	51,308
mtrt	Two-thread raytracing algorithm	62	629	24,435
jack	Java parser generator	86	743	36,253
SPECjbb2000	simulated transaction processing [25]	132	1778	73,608

Table 1. Benchmark characteristics. For each benchmark, this table gives the number of classes loaded and the number of methods and bytecodes dynamically compiled. The statistics include both application code and library code loaded at runtime. The first seven rows comprise the suite of SPECjvm98 benchmarks [24].

shorter execution duration than many of the other benchmarks, small changes will result in a larger performance impact. Interestingly, jess also experienced a code size decrease in almost every case, therefore, we can conclude that context-sensitivity enabled better quality code in this benchmark.

On the other hand, when db experienced performance improvements they were grouped with code size increases. In this case, context sensitivity enabled more inlining, resulting in a larger, but often faster, executable.

The results of the hybrid context-sensitivity policies are interesting. The first hybrid policy, Parameterless Class Methods, resulted in the most stable performance (nearly always within 1% of context-insensitive inlining), and the code size was similarly stable (less than 20% reduction.) The second hybrid policy, Parameterless Large Methods, resulted in much more dramatic behavior, yet is one of the few policies that resulted in a speedup on average.

The fact that the best speedup values for each benchmarks resulted from differing levels of context sensitivity in the non-adaptive policy confirms the intuition that no fixed level of context sensitivity is best for all call sites, and motivates the need for adaptive policies. The most notable observation from the runtime performance of the adaptive policies (Figures 4b–4f) is that overall benefits can be observed as compared to the non-adaptive policy.

Figure 6 depicts the average percentage of execution time spent in each component of the adaptive optimization system using the various profiling policies.[6] As we compare each of our policies to the baseline model (context-insensitive inlining – shown on the far left), we notice that the main difference is a significant (8-33%) reduction in the percentage of execution time devoted to optimizing compilation. By enabling a more focused set of inlining decisions, context-sensitive profiling eliminates useless inlining and thus significantly reduces compile time. It is also important to note that overhead of collecting and processing context-sensitive profile data is negligible. Although in some cases the context-sensitive system spends twice as much time in the AI Organizer and AOS listeners than the context-insensitive system, this overhead still represents less than 0.06% of total execution time.

In summary, context-sensitive profiling can significantly reduce optimized code space and compilation time while maintaining comparable runtime performance.

6. Related Work

A number of previous systems have used profile information to guide inlining decisions. In a typical system, context-insensitive profile data is gathered offline during a separate training run by executing an instrumented version of the application on a (hopefully) representative input set. The profile data is then fed into an optimizing compiler or whole program optimization system and used to identify the most desirable call sites to inline. Prototypical examples of systems using offline, context-insensitive profile data to guide inlining decisions include [20, 17, 8, 5, 2]. Offline systems can be quite effective, but are usually somewhat cumbersome to use and can be vulnerable to mispredictions due to variations in program behavior between the training and production runs of the application.

Most high performance Java virtual machines use online context-insensitive profile data to make inlining decisions. Previous work in Jikes RVM [3] demonstrated that online sampling of call edges could be used to drive profile-directed inlining, resulting in average speedups of 11%. Our work extends this system by sampling call traces instead of call edges. Suganuma et al. studied profile-directed inlining in the IBM DK [23]. Their system gathers context-insensitive profile information by temporarily interposing instrumentation code between the caller and callee at hot call sites that

[6] The remaining percentages are a combination of actual program execution time and garbage collection.

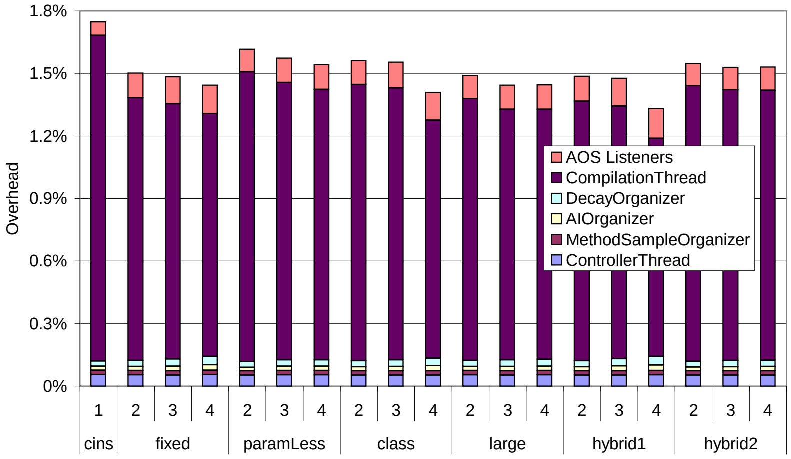


Figure 6. Percent of execution time spent in each component of the adaptive optimization system. The x-axis indicates the profiling policy used, along with maximum context-sensitivity depth. The y-axis indicates the percentage of time devoted to each component during execution. The baseline comparison value is located on the left, labeled *cins* for context-insensitive profiling.

are potential inlining candidates. The HotSpot^{TM}virtual machine gathers context-insensitive receiver class distributions during the initial interpretation of a method [18]. Unlike Jikes RVM and the IBM DK, HotSpot only gathers call edge data during initial interpretation of a method and thus is vulnerable to mispredictions if the application's calling patterns are different during initialization and steady state execution.

The Vortex optimizing compiler used offline context-sensitive profile data to guide its inlining decisions [13]. Vortex demonstrated that making inlining decisions using context-sensitive profile data can be quite valuable in optimizing object-oriented programs: context-sensitive profile data enabled speedups of up to 24% over context-insensitive profile data for large Cecil programs. These results were the inspiration for our work on adaptive context-sensitive inlining for Java programs. Like our system, Vortex's context sensitivity was based on call chains. However, its context-sensitivity policy was ad hoc: Vortex profiles optimized code and exploits this fact to tag profile data with the context information available from previous inlining. Vortex also benefits from being an offline system: profile data can be post processed to remove useless context sensitivity and filter out low probability edges.

The Self-93 system used online context-sensitive profile data for inlining [15, 14]. Like Vortex, the context sensitivity in Self-93 was due to profiling optimized code, not to an explicit context-sensitivity policy. The main differences between Self-93 and our work are: context sensitivity in Self-93 was ad hoc rather than managed by explicit adaptive polices, Self-93 gathered profile data via instrumentation in the PIC dispatching code rather than via call stack sampling, and no empirical assessment of the importance of context sensitivity for inlining was performed.

Call chain based context sensitivity is a very common practice in program analysis [21, 22]. Although context-sensitive profiling is less common, previous work has described several data structures for storing context-sensitive profile data. Vortex introduced a call chain representation for storing receiver class distributions that was optimized for the kinds of queries made by its inlining algorithm [13]. Ammons et al. developed the calling-context tree as a compact representation for context-sensitive profile data [1]. Later work by Whaley [26] and Arnold and Sweeney [4] described algorithms for constructing partial or approximate calling-context trees by periodic stack sampling.

Our system currently uses a very simple trace representation for its profile data. So far, this has been adequate but we are considering moving to a more sophisticated representation of the profile data that more efficiently supports the operations of the inlining organizer and inline oracle.

7. Conclusions

In this paper, we evaluated the use of adaptive, context-sensitive profile information for improving online method inlining. We described the implementation details and issues encountered as we incorporated our ideas into the Jikes RVM adaptive optimization system. We also provided details about several adaptive heuristics which approximate the *ideal* amount of context sensitivity, but in a manner lightweight enough to be used online in a virtual machine.

Our techniques were evaluated using various criteria. We assess the impact on overall runtime performance, code size, and the raw overhead of our implementation on the entire system. Our techniques were not simulated; they were implemented in an actual virtual machine, thus our performance numbers are actual wall-clock speedups that were achieved with a minimal amount of system tuning. On average, we found that with minimal impact on performance (+/-1%) context sensitivity can enable 10% reductions in compiled code space and compile time. Performance on individual programs varied from -4.2% to 5.3% while reductions in compile time and code space of up to 33.0% and 56.7% respectively were obtained.

We have also discovered that we have only begun to recognize the potential of adaptive context-sensitive inlining in an online setting. As our techniques mature and we investigate larger and more object-oriented programs, we expect context-sensitive inlining to be even more effective.

Acknowledgements

We would like to thank all of the contributors to Jikes RVM for helping to build the infrastructure that enabled this study. We thank Mike Hind and Vivek Sarkar for their input and support of this research. We also thank Mike Smith for his support of Kim Hazelwood's involvement. This research was performed while Kim was a summer intern at Watson.

References

[1] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation (PLDI)*, pages 85–96, Las Vegas, Nevada, 15–18 June 1997.

[2] M. Arnold, S. Fink, V. Sarkar, and P. Sweeney. A comparative study of static and dynamic heuristics for inlining. In *ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, Jan. 2000.

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

[4] M. Arnold and P. F. Sweeney. Approximating the calling context tree via sampling. Technical Report RC 21789, IBM T.J. Watson Research Center, July 2000.

[5] A. Ayers, R. Gottlieb, and R. Schooler. Aggressive inlining. In *Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation (PLDI)*, pages 134–145, Las Vegas, Nevada, 15–18 June 1997.

[6] B. Calder and D. Grunwald. Reducing indirect function call overhead in C++ programs. In *21st Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 397–408, Portland, OR, Jan. 1994.

[7] C. Chambers and D. Ungar. Iterative type analysis and extended message splitting: Optimizating dynamically-typed object-oriented programs. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 150–164, 1990.

[8] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W. mei W. Hwu. Profile-guided automatic inline expansion for C programs. *Software – Practice and Experience*, 22(5):349–369, May 1992.

[9] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *9th European Conf. on Object-Oriented Programming*, 1995.

[10] D. Detlefs and O. Agesen. Inlining of virtual methods. In *13th European Conference on Object-Oriented Programming*, June 1999.

[11] A. Diwan, J. E. Moss, and K. S. McKinley. Simple and effective analysis of statically-typed object-oriented programs. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 292–305, Oct. 1996.

[12] M. F. Fernandez. Simple and effective link-time optimization of Modula-3 programs. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 103–115, La Jolla, California, 18–21 June 1995.

[13] D. Grove, J. Dean, C. Garrett, and C. Chambers. Profile-guided receiver class prediction. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 108–123, Oct. 1995.

[14] U. Hölzle. *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming*. PhD thesis, Stanford University, Aug. 1994.

[15] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 326–336, June 1994. *SIGPLAN Notices, 29(6)*.

[16] R. Johnson. TS: An optimizing compiler for Smalltalk. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 18–26, 1988.

[17] W. mei W. Hwu and P. P. Chang. Inline function expansion for compiling C programs. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, volume 24, pages 246–255, June 1989. *SIGPLAN Notices* 24(6).

[18] M. Paleczny, C. Vick, and C. Click. The Java Hotspot(tm) Server Compiler. In *USENIX Java Virtual Machine Research and Technology Symposium*, pages 1 – 12, Apr 2001.

[19] J. Plevyak and A. A. Chien. Precise concrete type inference for object oriented languages. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 324–340, Oct. 1994.

[20] R. W. Scheifler. An analysis of inline substitution for a structured programming language. *CACM*, 20(9), Sep 1977.

[21] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, 1981.

[22] O. Shivers. Control flow analysis in Scheme. In *SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 164–174, June 1988.

[23] T. Suganuma, T. Yasue, and T. Nakatani. An empirical study of method inlining for a java just-in-time compiler. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM'02)*, July 2002.

[24] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98, 1998.

[25] The Standard Performance Evaluation Corporation. SPEC JBB 2000. http://www.spec.org/osg/jbb2000, 2000.

[26] J. Whaley. A portable sampling-based profiler for Java virtual machines. In *2000 Java Grande Conference*, June 2000.