A Framework for Persistence-Enabled Optimization of Java Object Stores

David Whitlock and Antony L. Hosking

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907-1398
USA
hosking@cs.purdue.edu
davidw@gemstone.com
http://www.cs.purdue.edu/~hosking

Abstract. Aggressive optimization of programs often relies on analysis and transformation that cuts across the natural abstraction boundaries of the source programming language, such as procedures in procedural languages, or classes in class-based object-oriented languages like Java. Unfortunately, execution environments for languages such as Java dynamically link code into the application as it executes, precluding cross-cutting analyses and optimizations that are too expensive to apply on-line.
Fortunately, persistent object systems usually treat the code base as an integral part of the persistent store. This code base approximates the notion of "whole-program" that has been exploited in other optimization frameworks. This paper describes an analysis and optimization framework for Java that operates against the persistent code base, and couples the results of analysis and optimization with the run-time system to ensure continued correctness of the resulting code. The framework performs extensive analysis over the code in the store, supporting optimizations that cut across class boundaries in ways that are not safe to perform off-line on stand-alone Java classes.

1 Introduction

Techniques for aggressive optimization of programs often rely on analyses and transformations that cut across the natural abstraction boundaries that allow for separate compilation of the source programming language. These atomic compilation units typically correspond to natural encapsulation boundaries, such as procedures in procedural languages, or classes in class-based object-oriented languages like Java. Such aggressive analyses and optimizations then take into account the particular combination of units that make up a given application program, and specialize the code from each unit to that particular combination. For statically-linked languages such as C and Modula-3, where the units that make up a given program are known statically, aggressive (and possibly expensive) analysis and optimization can be performed off-line. Unfortunately, execution environments for more dynamic languages such as Java link code into the application as it executes, precluding analyses and optimizations that may prove too expensive to apply

G.N.C. Kirby, A. Dearle, and D.I.K. Sjøberg (Eds.): POS-9, LNCS 2135, pp. 4–17, 2001.

on-line. Moreover, even for analyses that are not too expensive to apply on-line, some candidate optimizing transformations that are safe to apply at one time may subsequently become invalid if the code base evolves in a way that violates their safety.

Fortunately, persistent object systems usually treat the code base as an integral part of the persistent store. For example, the PJama prototype of orthogonal persistence for Java captures all classes loaded by a persistent application and stores them in the persistent store. This code base approximates the notion of "whole-program" that has been exploited in other optimization frameworks. We describe an analysis and optimization framework that operates against the code base of the PJama persistent store, and which couples the results of analysis and optimization with PJama's run-time system to ensure continued correctness of the resulting code. Our framework performs extensive analysis over the code in the persistent store, supporting optimizations that cut across class boundaries in ways that could not be safely achieved with stand-alone Java classes off-line.

The object-oriented programming paradigm aims to provide modular and reusable programs and libraries by encouraging the use of polymorphism and numerous small methods. However, the same qualities that make object-oriented programming attractive also make it difficult to optimize. Optimization techniques for procedural languages rely on potentially expensive static analysis of large portions of code. In contrast, object-oriented programs tend to be dynamic in nature and to have many short methods, so inlining is often employed to increase the size of the code regions for optimization.

We have implemented a number of analyses that attempt to reduce the dynamic nature of Java programs and, through method inlining, increase method size so that traditional optimizations may have a greater effect. The Java Language Specification [Gosling et al. 1996] requires that changes made to Java classes are *binary compatible* with pre-existing class binaries. However our optimizations break Java's data encapsulation model by allowing caller methods to access the private data of classes whose methods are inlined. For example, consider a method m_1 declared in class A and a method m_2 declared in class B that accesses one of B's private fields. Inlining a call in m_1 to m_2 would result in m_1 accessing one of B's private fields. This violates encapsulation as well as Java's static rules that ensure binary compatibility. Thus, our optimizations must be performed on classes in a safe environment in which the restrictions placed on transformations to ensure binary compatibility can be lifted. Code could be optimized at runtime when the virtual machine has complete control over the classes. However, our optimizations require extensive program analysis whose runtime cost would most likely outweigh any benefit gained by optimization.

Some implementations of orthogonal persistence for Java maintain a representation of classes, as well as instantiated objects, in the persistent store. Such a store provides a good approximation of the closed-world environment in which a Java program may actually run. The classes in a persistent store are verified to be binary compatible upon their entry to the virtual machine, so there is no need to "reverify" classes in the persistent store. Moreover, a program executing within a persistent store has an unusual concept of "runtime". Because data persists between executions in its runtime format, the execution of the program can be thought of in terms of the lifetime of its data. The program runs, pauses (no code executes, but the runtime data persists), then resumes. When the

program is "paused" classes within the store may be modified without regard to binary compatibility. It is during these "pauses" that we perform our optimizations on classes residing with an persistent store.

Because our optimizations are static they cannot account for new classes that are introduced via class reflection or native methods. The introduction of new classes may invalidate some of our optimizations. To handle this, the virtual machine notifies the optimizer at runtime of changes in the class hierarchy. The optimizer, in turn, may revert certain methods to their unoptimized form, de-optimizing them based on the information provided by the virtual machine. We use a property known as *pre-existence* (discussed below) to avoid the need to de-optimize active methods (i.e., to avoid on-stack replacement).

Here we explore performing extensive *off-line* optimization of Java classes that reside in a persistent store. By off-line, we do not necessarily mean that the persistent system is inactive – transformations may be performed concurrently with the persistent application – but rather that the type analyses necessary for the optimizing transformations must not be invalidated by the concurrent execution through dynamic class loading. Thus, one might envision the optimizer running in a background thread, taking advantage of quiescent periods in the store. For now, however, our implementation operates against the persistent store as a separate, privileged application. This is unavoidable since we use the PJama prototype of orthogonal persistence for Java [Atkinson et al. 1996], which currently has no support for controlled concurrent access to a store by multiple applications. We use our Bytecode-Level Optimizer and Analysis Tool (BLOAT) [Hosking et al. 1999; Hosking et al. 2000] to model, analyze, and optimize Java programs, with method inlining to expose additional optimization opportunities to our intra-procedural optimizations.

2 Type Analysis

A *callee* method is invoked by a *caller* method at a *call-site*. There are two kinds of call-sites. A *dynamically bound* call-site invokes an *instance method* and requires a run-time *dynamic method lookup* to determine the exact method to be invoked based on the type of its *receiver* object. A *statically bound* call-site invokes a *class method* or a *constructor method* and does not require such a lookup. Type analysis computes the possible types of a receiver object and thus computes a set of methods that could be invoked at a call-site. If only one method can be invoked at a call-site, the call-site is said to be *monomorphic*. Otherwise, it is *polymorphic*.

Class hierarchy analysis [Dean et al. 1995; Fernandez 1995; Diwan et al. 1996] uses the class inheritance hierarchy in conjunction with static type information about a call-site to compute the possible methods that may be invoked. We use the class hierarchy given in Figure 1 to demonstrate our analyses and optimizations. Consider the getArea method in Figure 2 which contains a dynamically bound call to the area method. Without considering type and flow analysis (i.e., ignoring the context of the call), then the call to area might resolve to the implementation of the area method in any of the Triangle, Circle, or Rectangle classes. Observe that if the compile-time type of the receiver of the invocation of area is Rectangle or its subclass Square, the only method that could

```
abstract class Shape {
  abstract float area();
  float getPI() { return 3.14579F; }
  static float square(float f) {
    return f * f;
  }
}

class Triangle extends Shape {
  float b, h;
  Triangle(float b, float h) {
    this.b = b; this.h = h;
  }
  float area() { return b * h / 2; }
}

class Circle extends Shape {
  float r;
  Circle(float r) { this.r = r; }
  float area() {
     return getPI() * Shape.square(r);
  }
}

class Rectangle extends Shape {
  float s1, s2;
  Rectangle(float s1, float s2) {
    this.s1 = s1; this.s2 = s2;
  }
  float area() { return s1 * s2; }
}

class Square extends Rectangle {
  Square(float s) { super(s, s); }
}
```

Fig. 1. Example class hierarchy

possible be invoked is the area implementation in Rectangle because no subclass of Rectangle overrides area. This observation is key to class hierarchy analysis.

Rapid type analysis (RTA) [Bacon and Sweeney 1996] extends class hierarchy analysis by using class instantiation information to reduce the set of potential receiver types at a call-site. Consider the program in Figure 2. Class hierarchy analysis stated that the call to area could invoke the area method of Triangle, Circle, or Rectangle. However, a quick glance at the program reveals that it is impossible for the area method implemented in Triangle to be invoked because neither Triangle nor any of its subclasses is instantiated. Classes that are instantiated are said to be *live*.

RTA must be careful in the way that it marks a class as being instantiated. Invoking a class's constructor does not necessarily mean that the class is instantiated. Consider an invocation of the one-argument constructor of Square. Calling this constructor indicates that class Square is instantiated. However, Square's constructor invokes the constructor of its superclass, Rectangle. This invocation of Rectangle's constructor does not indicate that Rectangle is instantiated.

```
float getArea(boolean b) {
  Shape s;
  if(b)
    s = new Circle(2);
  else
    s = new Square(3);
  float area = s.area();
  return area;
}
```

Fig. 2. Calling method area

Rapid type analysis examines a program's methods starting at its entry point (e.g., main method). One following operations occurs at an invocation of method m.

- If the call-site is statically bound, then m is marked as being live and is examined further.
- If the call-site is dynamically bound, then the set of potential receiver types is calculated using class hierarchy analysis. For each potential receiver type that has been instantiated, T, the implementation of m that would be invoked with receiver type T is made live and is examined further. The implementations of m in the uninstantiated classes are "blocked" on each uninstantiated type T.
- If m is a constructor of class T and the caller is not a constructor of a subclass of T, the class T is instantiated and m becomes live and is examined. Additionally, any methods that were blocked on T are unblocked and examined.

In our running example, classes Circle and Square are live. The constructors for Circle and Square, the area methods of Circle and Rectangle, and the getPI and square methods are all live. The area method of Triangle is blocked on Triangle.

3 Call-Site Customization

The compiler for the SELF language introduced the notion of call-site *customization* [Chambers et al. 1989]. Customization optimizes a dynamically bound call-site based on the type of the receiver. If the type of the receiver can be precisely determined during compilation, then the call-site can be statically bound.

The call to area in Figure 2 can be customized in the following manner. Rapid type analysis concluded that the the area method of either Circle or Rectangle will be invoked. Customization replaces the dynamically bound call-site with two type tests and corresponding statically bound invocations (in the form of a call to a class method) as show in Figure 3. If the call-site is monomorphic, no type test is necessary. Two class methods, $area in Circle and $area in Rectangle have been created containing the same code as the instance method versions of area. In the case that the receiver type is none of the expected types, the virtual method is executed.

Once a call to an instance method has been converted into a call to a class method, the call may be *inlined*. Inlining consists of copying the code from the callee method into the caller method. Thus, inlining completely eliminates any overhead associated with invoking the method.

```
float getArea(boolean b) {
  Shape s;
  if (b)
    s = new Circle(2);
  else
    s = new Square(3);
  float area;
  if (s instanceof Circle) {
    Circle c = (Circle) s;
    area = Circle.$area(c);
  } else if(s instanceof Rectangle) {
    Rectangle r = (Rectangle) s;
    area = Rectangle.$area(r);
  } else
    area = s.area();
  return area;
}
```

Fig. 3. Customized call to area

Our optimization framework performs intra-procedural data-flow analysis that in some cases precisely determines the types of receiver objects. For instance, if the receiver object is created within the caller method by a constructor call, we know the receiver's type.

Our analyses have resulted in an increased number of statically bound call-sites whose callee methods are precisely known and may be *inlined*. Inlining involves copying the callee's code into the caller method. By inlining methods, we not only eliminate the overhead of invoking a method, but we also give our intra-procedural optimizer a large code context in which to perform its optimizations.

4 Pre-existence

When customizing call-sites certain assumptions are made about the classes in the program. For instance, the analysis may determine that a call-site is monomorphic and inlines the invocation. However, additional classes may enter the system that invalidate assumptions made during the analysis. In this case the optimized code must be de-optimized.

This situation is further exacerbated by the fact that optimized code may need to be de-optimized while it is executing. Consider the program in Figure 4a. The method getSomeShape may potentially load a subclass of Shape that is unknown at analysis time. getSomeShape could be a native method or, in the worst case, could ask the user for the name of a class to load. In any case, the call to area cannot be inlined without the possibility of later adjustment.

The SELF system [Hölzle et al. 1992] solved this problem by using a run-time mechanism called *on-stack replacement* to modify executing code. SELF maintains a significant amount of debugging information that allows for quick de-optimization of optimized code. When an optimization is invalidated, SELF recovers the original source code and re-optimizes it taking the invalidating information into account. Maintaining the amount of information necessary to perform these kinds of optimizations requires a noticeable

```
float getSomeArea() {
  Shape s = getSomeShape();
  float area = s.area();
  return area;
}
```

(a) Receiver does not pre-exist

```
float getSomeArea(Shape s) {
  float area = s.area();
  return area;
}
```

(b) Receiver pre-exists

Fig. 4. Pre-existence of receiver objects

space and time overhead, increases the complexity of the optimizer, and places certain constraints on the kinds of optimizations that can be performed.

Detlefs and Agesen [1999] introduced the concept of *pre-existence* to eliminate the need for on-stack replacement. Consider a method foo containing an invocation of method bar with receiver object o. o is said to *pre-exist* if it is created before foo is called. The type of any pre-existent object must have been introduced before the method foo is called. Any invalidation of assumptions made about the type of o this introduction may cause will not affect the method foo while it is executing. Therefore, on-stack replacement on method foo will never occur and it is safe to inline the call to bar.

Consider the version of getSomeArea presented in Figure 4b. In this method the call-site's receiver is one of the method's arguments. If any previously unknown subclass of Shape were to enter the system, it would have to do so before the call to getSomeArea. At the time that the new class enters the system, the as-yet-uncalled getSomeArea method would be appropriately re-optimized to account for the new class.

A technique called *invariant argument analysis* is used to determine whether or not a call-site's receiver pre-exists. Invariant argument analysis traces the uses of the caller's arguments. If an argument is used as receiver, then it pre-exists. Additionally, receivers that result solely from allocations operations pre-exist.

5 Implementation

We analyze and optimize *class files*, a binary representation of Java classes that are suitable for execution on a Java virtual machine, and are able to take advantage of the semantics of some of the machine's instructions. When performing rapid type analysis, instead of analyzing calls to constructors, we use occurrences of the new instruction to denote a class instantiation. When customizing call-sites we transform invokevirtual instructions which call an instance bound method to invokespecial instructions that invoke an instance method, but do not perform a dynamic method lookup.

To accommodate our optimizations several changes were made to the PJama virtual machine. By allowing caller methods access to the private data of their inlined methods, our optimizations break encapsulation. Once a class is loaded into the virtual machine, it is no longer necessary to enforce encapsulation. Thus, we disable data access checks for loaded classes. Also, for security reasons, the Java Virtual Machine Specification [Lindholm and Yellin 1999] allows the invokespecial instruction to only invoke superclass and private methods. We have relaxed this rule and allow invokespecial to call any instance method.

6 De-optimization

Pre-existence ensured that classes entering the system could never invalidate executing code. However, other optimized code may need to be de-optimized in the face of such changes to the type system. Consider a caller method foo that contains a call to the area method. Suppose that rapid type analysis has determined that the call-site will only invoke the area method of Triangle and that its receiver object pre-exists because it is a method parameter. Customization will transform this call into a non-virtual call to the area method of Triangle. Suppose further that at runtime the program, using Java's reflection mechanism, loads the EquilateralTriangle class, a subclass of Triangle that overrides the area method. During customization we assumed that no subclass of Triangle overrode the area method. However, the introduction of the EquilateralTriangle invalidates this assumption and the customized invocation of the area method of Triangle is incorrect because the receiver object may be an instance of EquilateralTriangle in addition to Triangle. Thus, we must de-optimize foo at runtime by undoing the effects of customization. In an attempt to make de-optimization as fast as possible, foo is simply reverted to its unoptimized form.

In the above example, we say that method foo *depends* on the area method of Triangle because if the area method of Triangle is overridden, then foo must be de-optimized. The optimizer maintains a series of dependencies [Chambers et al. 1995] among methods resulting from call-site customization. As a result of the customization shown in Figure 3 the getArea method would depend on the area method of Circle and the area method of Rectangle. Note that if our analysis can precisely determine the type(s) of a receiver object (e.g., the object is created inside the caller method), then no dependence is necessary. The dependencies are represented as Java objects and reside in the persistent store.

The PJama virtual machine was modified to communicate with the optimizer at runtime to determine when methods should be de-optimized. When a class is loaded into the virtual machine, the optimizer is notified. If the newly-loaded class invalidates any assumptions made about the class hierarchy during optimization, the optimizer consults the method dependencies and de-optimizes the appropriate methods. To account for any degradation in performance that de-optimization may produce, it may be desirable to re-optimize a Java program multiple times during its lifetime. Subsequent re-optimizations will account for classes that are introduced by reflection.

A persistent store provides a closed-world model of a Java program, allows us to disregard the restriction of binary compatibility, and provides a repository in which the optimizer can store data necessary for de-optimization to ensure correct program behavior when classes are introduced into the system at runtime. Thus, persistence enables us to safely perform our inter-procedural optimizations on Java programs.

It is important to note that the general cross-class optimizations we support cannot be performed outside the virtual machine by modifying the bytecode at load time, since the modified bytecode would violate Java's structural constraints and fail bytecode verification.

Table 1. Benchmarks

Name	Description
crypt	Java implementation of the Unix crypt utility
db	Operations on memory-resident database
huffman	Huffman encoding
idea	File encryption tool
jack	Parser generator
jess	Expert system
jlex	Scanner generator
jtb	Abstract syntax tree builder
lzw	Lempel-Ziv-Welch file compression utility
mpegaudio	MPEG Layer-3 decoder
neural	Neural network simulation

7 Results

To evaluate the impact of inter-procedural optimizations on Java programs, we optimized several Java benchmark applications and compared their performance using static and dynamic performance metrics. To obtain the measurements we used a software library that allows user-level access to the UltraSPARC hardware execution counters,[1] permitting us to gain accurate counts of hardware clock cycles, cache misses such as instruction fetch stalls and data read misses. Benchmarks were run with the operating system in single-user mode to avoid spurious interference from unrelated processes.

The experiments were performed on a Sun Ultra 5 with a 333 MHz UltraSPARC-IIi processor with a 2MB external (L2) cache and 128MB of RAM. The UltraSPARC-IIi has a 16-KB write-through, non-allocating, direct mapped primary data cache that is virtually-indexed and virtually-tagged. The 16-KB primary instruction cache is two-way set associative, physically indexed and tagged, and performs in-cache 2-bit branch prediction with single cycle branch following.

We used eleven benchmarks programs as described in Table 1 to measure the impact of inter-procedural optimizations. Several of the benchmarks were taken from the SpecJVM [SPEC 1998] suite of benchmarks. Table 2 gives some static statistics about the benchmarks: the number of live classes and methods, the number of virtual call-sites, the percentage of those call-sites that pre-exist, and the percentage of pre-existent call-sites that are monomorphic and duomorphic (only two methods could be invoked). It is interesting to note that for most benchmarks the majority of virtual call-sites are precluded from inlining because they do not pre-exist. Note also that nearly all (89.8–95.7%) of pre-existent call-sites are monomorphic or duomorphic. Thus, from a static point of view, extensive customization of polymorphic call-sites seems unnecessary.

Table 3 summarizes the time spent optimizing each benchmark and supports our claim that our inter-procedural optimizations are too expensive to perform during program execution. The optimizer spends the vast majority of its time constructing the call

[1] See http://www.cs.msu.edu/∼enbody/

Table 2. Inlining statistics (static)

Benchmark	live classes	live methods	virtual calls	% preexist	% mono	% duo
crypt	134	853	1005	38.9	87.0	3.1
db	151	1010	1373	36.7	87.7	4.2
huffman	141	875	1071	38.6	87.7	2.9
jack	184	1170	2305	31.5	86.1	8.3
jess	245	1430	2563	35.0	92.2	3.0
jlex	154	1008	1315	35.4	88.8	2.6
jtb	273	2111	3965	32.8	87.7	8.0
lzw	142	905	1031	38.3	86.8	3.0
mpegaudio	173	1146	1594	31.6	87.1	5.2
neural	139	883	1024	39.1	87.2	3.0

Table 3. Optimization times

Benchmark	Total (sec)		% Call Graph		% Customize		% Inline		% Commit	
	User	System	User	System	User	System	User	System	User	System
crypt	75.40	17.63	56.84	81.79	7.88	0.62	19.58	0.51	15.70	17.07
db	317.97	61.30	74.68	84.34	5.99	0.20	7.30	1.55	12.03	13.92
huffman	78.59	18.29	58.65	82.23	7.79	0.55	18.59	0.71	14.98	16.51
jack	333.69	65.56	71.34	81.18	6.54	0.23	9.85	5.54	12.27	13.06
jess	394.01	72.26	66.50	75.63	9.17	0.26	12.17	10.64	12.16	13.47
jlex	90.11	19.03	55.03	81.87	7.70	0.84	22.02	0.68	15.25	16.61
jtb	258.14	23.26	31.05	75.71	7.53	0.47	46.82	3.83	14.60	19.99
lzw	75.43	18.43	56.81	82.69	7.93	0.38	19.58	0.65	15.68	16.28
mpegaudio	351.02	64.04	74.09	85.81	7.17	0.27	7.69	0.50	11.04	13.43
neural	74.52	18.63	58.99	81.59	8.24	0.27	17.35	1.13	15.42	17.02

graph and committing the optimized classes back to class-files. Call-site customization and inlining is comparatively inexpensive to perform.

Each benchmark was optimized in five configurations: no optimization, (nop), only intra-procedural optimizations (intra), call-site customization (cust), inlining of non-virtual calls (inline), and intra-procedural optimizations on top of inlining (both). Our intra-procedural optimizations include dead code elimination, constant/copy propagation, partial redundancy elimination, and register allocation of Java virtual machine local variables. Through analysis of empirical data, several conditions on the inter-procedural optimizations were arrived at: only monomorphic call-sites were customized, no callee method that is larger than 50 instructions is inlined and no caller method is allowed to exceed 1000 instructions because of inlining.

Examining the number of bytecodes executed provides insight into the effectiveness of our inter-procedural optimizations. Figures 5 and 6 summarize bytecode counts for the five optimization levels: nop, intra, cust, inline, and both. As Figure 5 demonstrates

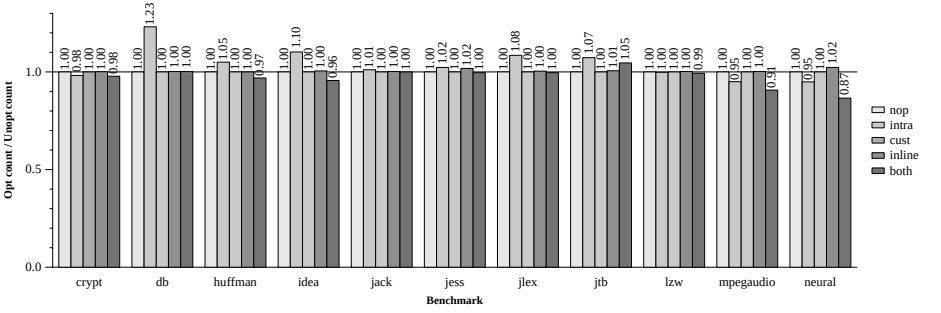

Fig. 5. Total bytecodes executed

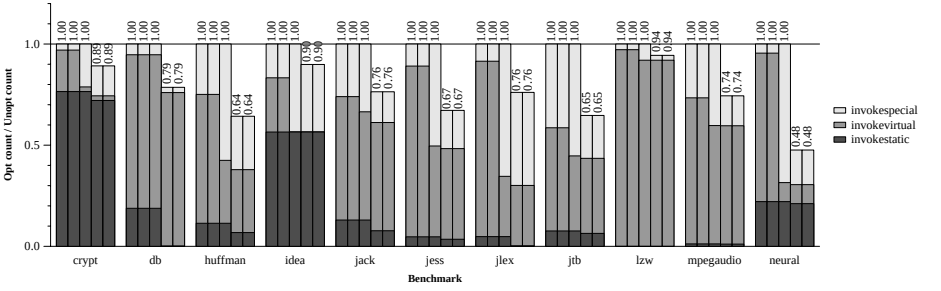

Fig. 6. Method invocations

the inter-procedural optimizations, cust and inline, do not have a significant effect on the total number of bytecodes executed. However, combining inter-procedural and intra-procedural optimizations (both) results in up to 8% fewer bytecodes being executed than with the intra-procedural optimizations alone (intra).

The effects of call-site customization and method inlining can be seen by examining the number and kind of methods executed. Figure 6 reports the number of invokespecial, invokevirtual[2], and invokestatic instructions. Call-site customization (cust) results in an often drastic reduction in the number of invokevirtual instructions. Likewise, method inlining removes as many as 52% of method invocations. For several benchmarks (crypt, idea, and neural) very few static method invocations are inlined. This is most likely due to the fact that the bodies of these methods exceed the 50 instruction limit placed on inlined methods.

Figure 7 compares the execution times (number of cycles executed) of our benchmarks. The benchmarks were executed on the Sun Java 2 SDK Solaris^{TM} Production Release Virtual Machine with Just-In-Time compilation disabled[3]. Our optimizations cause a -2–22% decrease in the number of machine cycles. For several benchmarks, our optimizations cause an increase in the number of instruction fetch stalls and data read misses leading to an increase in the number of cycles.

[2] There were a negligible number of invokeinterface instructions executed.

[3] For some benchmarks, our optimizations expose a bug in the JIT.

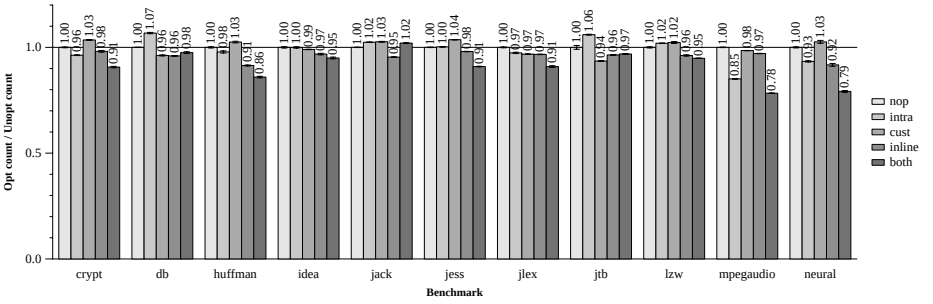

Fig. 7. Execution times (cycles)

For most benchmarks customizing monomorphic call-sites has little effect on the number of cycles executed. This leads us to believe that the interpreter's invokevirtual instruction has been optimized for maximum efficiency since it appears to have the same cost as the non-virtual invokespecial instruction. However, the increase in speed provided by method inlining demonstrates that the method invocation sequence is still costly. In most cases inlining enabled the intra-procedural optimizations to increase performance further.

8 Related Work

Much work has been done in the area of type analysis of object-oriented programming languages, particularly in type prediction and type inferencing. Palsberg and Schwartzbach [1991] present a constraint-based algorithm for inter-procedural type inferencing that operates in $O(n^3)$ time where n is the size of the program. Agesen [Agesen 1994] presents a survey of various improvements to the $O(n^3)$ algorithm. Several strategies create copies of methods called "templates" whose type information is specialized with respect to the type of the parameters. Agesen also describes the "Cartesian Product Algorithm" [Agesen 1995] that creates a template for every receiver and argument tuple per call-site. Several of the above algorithms were considered for our type analysis. However, as implementation began it became obvious that none of them is practical using our modeling framework for the numerous classes in JDK1.2.

Diwan et al. [1996] use class hierarchy analysis and an intra-procedural algorithm in addition to a context-insensitive type propagation algorithm to optimize Modula-3 programs. Budimlic and Kennedy present inter-procedural analyses and method inlining of Java programs [Budimlic and Kennedy 1997; Budimlic and Kennedy 1998]. They implement *code specialization* in which virtual methods contain a run-time type test to determine whether or not inlined code should be executed. In order to preserve Java's encapsulation mechanism, their analyses must operate on one class at a time. Thus, no method's from other classes may be inlined.

The Soot optimization framework [Sundaresan et al. 1999] performs similar analysis to ours. While they describe type analyses that are more aggressive than rapid type analysis, it is unclear as to the practicality of these analyses under JDK1.2.

The Jax application extractor [Tip et al. 1999] uses rapid type analysis to determine the essential portions of a Java program with the goal of reducing the overall size of the application. Jax performs several simple optimizations such as inlining certain accessor methods and marking non-overridden methods as being final and respects Java's data encapsulation rules. Unlike the other tools described above, Jax accounts for dynamic changes in the type system via a specification provided by the user.

Several studies [Grove et al. 1995; Fernandez 1995] examine the effects of using run-time profiling data to optimize object-oriented programs. Profiling data can be used to identify sections of code that are executed frequently where optimizations may have greater impact as well as the true types of the receivers of method calls.

More recently, the Jalepeño Java Virtual machine [Alpern et al. 1999] has taken a unique approach to optimizing Java program. Jalepeño is written almost entirely in Java and yet it executes without a bytecode interpreter. It employs several compilers that translate bytecode into native machine instructions. The compilers use both static techniques and profiling data, but differ in the number and kinds of optimizations they perform.

9 Conclusions

Java programs whose classes reside inside a persistent store give us a unique opportunity for whole-program optimization. We can relax certain constraints placed on stand-alone Java programs and safely perform expensive off-line optimizations. In order to ensure correct program execution in the face of an evolving system, certain optimizations are undone at runtime. Our results show that our optimizations are able to remove a significant portion of the dynamic call overhead associated with Java programs, and to inline methods for more effective optimization of the resulting regions of larger context.

Acknowledgements. This research is supported in part by the National Science Foundation under Grant No. CCR-9711673 and by gifts from Sun Microsystems and IBM.

References

AGESEN, O. 1994. *Constraint-Based Type Inference and Parametric Polymorphism*. lncs, vol. 864. 78–100.

AGESEN, O. 1995. The cartesian product algorithm: Simple and precise typing of parametric polymorphism. See ECOOP'95 [1995], 2–26.

ALPERN, B., ATTANASIO, C. R., BARTON, J. J., COCCHI, A., HUMMEL, S. F., LIEBER, D., NGO, T., MERGEN, M., SHEPHERD, J. C., AND SMITH, S. 1999. Implementing Jalapeño in Java. See OOPSLA'99 [1999], 314–324.

ATKINSON, M. P., DAYNÈS, L., JORDAN, M. J., PRINTEZIS, T., AND SPENCE, S. 1996. An orthogonally persistent Java. *ACM SIGMOD Record 25,* 4 (Dec.), 68–75.

BACON, D. F. AND SWEENEY, P. F. 1996. Fast static analysis of c++ virtual function calls. See OOPSLA'96 [1996], 324–341.

BUDIMLIC, Z. AND KENNEDY, K. 1997. Optimizing Java: Theory and practice. *Software—Practice and Experience 9,* 6 (June), 445–463.

BUDIMLIC, Z. AND KENNEDY, K. 1998. Static interprocedural optimizations in java. Tech. Rep. CRPC-TR98746, Rice University.

CHAMBERS, C., DEAN, J., AND GROVE, D. 1995. A framework for selective recompilation in the presence of complex intermodule dependencies. In *Proceedings of the International Conference on Software Engineering* (Seattle, Washington, Apr.). IEEE Computer Society, 221–230.

CHAMBERS, C., UNGAR, D., AND LEE, E. 1989. An efficient implementation of Self, a dynamically-typed object-oriented language based on prototypes. In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (New Orleans, Louisiana, Oct.). *ACM SIGPLAN Notices 24,* 10 (Oct.), 49–70.

DEAN, J., GROVE, D., AND CHAMBERS, C. 1995. Optimization of object-oriented programs using static class hierarchy analysis. See ECOOP'95 [1995].

DETLEFS, D. AND AGESEN, O. 1999. Inlining of virtual methods. In *Proceedings of the European Conference on Object-Oriented Programming* (Lisbon, Portugal, June), R. Guerraoui, Ed. Lecture Notes in Computer Science, vol. 1628. Springer-Verlag, 258–278.

DIWAN, A., MOSS, J. E. B., AND MCKINLEY, K. S. 1996. Simple and effective analysis of statically-typed object-oriented programs. See OOPSLA'96 [1996], 292–305.

ECOOP'95 1995. *Proceedings of the European Conference on Object-Oriented Programming* (Åarhus, Denmark, Aug.). Lecture Notes in Computer Science, vol. 952. Springer-Verlag.

FERNANDEZ, M. F. 1995. Simple and effective link-time optimization of Modula-3 programs. In Proceedings of the ACM Conference on Programming Language Design and Implementation (La Jolla, California, June). *ACM SIGPLAN Notices 30,* 6 (June), 103–115.

GOSLING, J., JOY, B., AND STEELE, G. 1996. *The Java Language Specification*. Addison-Wesley.

GROVE, D., DEAN, J., GARRETT, C., AND CHAMBERS, C. 1995. Profile-guided receiver class prediction. In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (Austin, Texas, Oct.). *ACM SIGPLAN Notices 30,* 10 (Oct.), 108–123.

HÖLZLE, U., CHAMBERS, C., AND UNGAR, D. 1992. Debugging optimized code with dynamic deoptimization. In Proceedings of the ACM Conference on Programming Language Design and Implementation (San Francisco, California, June). *ACM SIGPLAN Notices 27,* 7 (July), 32–43.

HOSKING, A. L., NYSTROM, N., WHITLOCK, D., CUTTS, Q., AND DIWAN, A. 1999. Partial redundancy elimination for access path expressions. In *Proceedings of the Intercontinental Workshop on Aliasing in Object Oriented Systems* (Lisbon, Portugal, June).

HOSKING, A. L., NYSTROM, N., WHITLOCK, D., CUTTS, Q., AND DIWAN, A. 2000. Partial redundancy elimination for access path expressions. *Software–Practice and Experience*. To appear in Special Issue on Aliasing in Object-Oriented Systems.

LINDHOLM, T. AND YELLIN, F. 1999. *The Java Virtual Machine Specification*. Addison-Wesley.

OOPSLA'96 1996. *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (San Jose, California, Oct.). *ACM SIGPLAN Notices 31,* 10 (Oct.).

OOPSLA'99 1999. *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Denver, Colorado, Nov.). *ACM SIGPLAN Notices 34,* 10 (Oct.).

PALSBERG, J. AND SCHWARTZBACH, M. I. 1991. Object-oriented type inference. In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (Phoenix, Arizona, Oct.). *ACM SIGPLAN Notices 26,* 11 (Nov.), 146–161.

SPEC. 1998. SPECjvm98 benchmarks. http://www.spec.org/osg/jvm98.

SUNDARESAN, V., RAZAFIMAHEFA, C., VALLÉE-RAI, R., AND HENDREN, L. 1999. Practical virtual method call resolution for java. Trusted objects, Centre Universitaire d'Informatique, University of Geneva. July.

TIP, F., LAFFRA, C., SWEENEY, P. F., AND STREETER, D. 1999. Practical experience with an application extractor for Java. See OOPSLA'99 [1999], 292–305.