PERSISTENCE-ENABLED OPTIMIZATION OF JAVA PROGRAMS

A Thesis

Submitted to the Faculty

of

Purdue University

by

David Michael Whitlock

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

May 2000

To my Grandpa Granger, although I doubt much of this would have made sense to him.
Come to think of it, we always had a lot in common.

ACKNOWLEDGMENTS

Thanks to my family back in New Hampshire for all of their moral support and encouragement. Thanks to my advisor, Tony Hosking, for providing me with the opportunity to work with BLOAT. Many thanks to the gang in the S^3 lab for putting up with me while I was writing. Thanks to Steve, Quintin, and Stuart in Glasgow for being great lads. Thanks to the PJama group at Sun, especially Brian Lewis, for all of their help. And lastly, my gratitude to Nate Nystrom for conceiving and writing BLOAT. I still don't know how he did it all.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

ABSTRACT

Whitlock, David Michael. M.S., Purdue University, May 2000. Persistence-Enabled Optimization of Java Programs. Major Professor: Antony Hosking.

Programs contained within a persistent store provide a unique opportunity for whole-program analysis and optimization. In particular, several constraints on optimization can be relaxed in a persistent setting, allowing aggressive off-line optimizations to be performed. This work explores interprocedural optimizations of Java programs residing within a persistent store. Call site customization and method inlining are safely performed on Java methods without the need to adjust methods while they execute. A feedback mechanism between the Java virtual machine and the optimizer allows new classes to be added to the system while maintaining correctly optimized code. Our results show a significant decrease in the number of method invocations and a noticeable increase in performance in certain execution environments.

1 INTRODUCTION

Because of its dynamicism, object-orientation, and platform independence the Java™ programming language has become a popular language choice for application programming. However, the same features that make the Java language popular also make it difficult to optimize. Unlike traditional procedural programming languages, complete whole-program analysis cannot be performed on Java programs. The object-oriented programming style encourages the use of small methods that cannot be extensively optimized. Additionally, the Java execution model places constraints on the types of program optimizations that can be performed.

1.1 Interprocedural Analysis

Much work has been done in the area of interprocedural analysis and optimizations for object-oriented programming languages. Method inlining is a popular and effective optimization that not only removes the overhead of method invocation, but also provides the optimizer with a larger code context in which to optimize. Unfortunately, interprocedural analyses are often expensive and complex and require that certain assumptions about the program be made. However, at runtime additional information that is not available during the analysis may enter the system and invalidate certain optimizations. The invalidation and subsequent “deoptimization” of the optimized code is essential to the correct execution of a program.

1.2 Orthogonal Persistence

This work presents a novel approach to whole-program optimization that performs extensive off-line analysis and optimization of Java programs. The analysis and optimization data, as well as the program being optimized, are maintained in a persistent store. At runtime, operations that invalidate the optimizations can be detected. The data in the store is then consulted and the optimized code is corrected. By coupling the optimizer and the runtime system, significant off-line analysis can occur while guaranteeing safe, correct program execution.

1.3 Measurements

We applied our interprocedural optimizations to a number of Java programs. Static and dynamic measurements such as number and kind of bytecodes executed, elapsed time, and hardware cache behavior were taken. The results demonstrate the ability of our optimizations to reduce significantly the number of methods that are invoked and to reduce the overhead of many of the invocations that remain. Our results also demonstrate that intraprocedural optimizations do indeed take advantage of the larger context provided by interprocedural optimizations and impact performance further.

1.4 Overview

The rest of this thesis is organized as follows. In Chapter 2 we discuss the interprocedural analysis and optimizations we perform on Java programs. Chapter 3 describes the implementation of these optimizations using the Bytecode-Level Optimizations and Analysis Tools [Nystrom 1998] optimization framework. Chapter 4 outlines the experimental methodology used to measure the performance of optimized programs and presents the results of those experiments. We conclude with a discussion of related work and directions for future work.

2 BACKGROUND

2.1 The Java Object Model

The object-oriented programming paradigm aims to provide modular and reusable programs and libraries through data and code encapsulation. The object model for the Java programming language is described in Gosling et al. [1996]. A *type* is associated with program entities such as variables and expressions. A type limits the values an entity may hold and defines the operations the entity must provide. A *class* defines a new type and describes how it is implemented. An *object* is a class instance or an array. At compile-time, we say that an object "has a type" where at run-time we say that it "belongs to a class". Classes that are declared to be abstract cannot be instantiated. An *interface* declaration specifies a type consisting of constants and abstract methods.

With the exception of the java.lang.Object class, all classes have a *direct superclass*. A class is said to be a *direct subclass* of its direct superclass. A direct subclass *derives* its implementation from its direct superclass. Classes that are marked final may not have subclasses. The derivation relationship among classes forms a *class hierarchy*.

A type S is *assignment compatible* with a type T according to the following rules:

- If S is a class type
  - If T is a class type, then S and T must be the same class or S must be a subclass of T
  - If T is a interface type, then S must implement T
- If S is an interface type

  - If T is a class type, it must be java.lang.Object
  - If T is an interface type, then S and T must be the same interface or S must be a subinterface of T

- If S is an array type SC[] whose components are of type SC
  - If T is a class type, then T must be java.lang.Object
  - If T is an interface type, then T must be java.lang.Cloneable or java.io.Serializable
  - If T is an array type TC[] whose components are of type TC, then TC must be assignment compatible to SC

A *method* is executable code that can be invoked and is associated with a class. A *class method* is invoked relative to a class type, while an *instance method* is invoked with respect to a class instance. Methods are invoked at a *call site*. The method containing the call site is referred to as the *caller* and the method that is invoked is referred to as the *callee*. Methods have a fixed number of formal parameters, each of which has a type, and may return a value to the caller. A method's signature consists of the name of the method and the number and types of the method's formal parameters. A *constructor* is executable code that initializes class instances. Unlike methods, constructors cannot be invoked directly. *Native* methods are implemented in platform-dependent code.

A class that *directly implements* an interface has an implementation of all the abstract methods specified by the interface. It is possible that a class may declare a method with the same signature as a method in its superclass. If the method is an instance method, we say that the method *overrides* the method of its superclass. If the method is a class method, we say that the method *hides* the method of the superclass. A class *inherits* the methods of its direct superclass and direct superinterfaces that are neither overridden nor hidden by a declaration in the class.

2.2 Methods

2.2.1 Call Site Binding

When a Java program is compiled, several steps determine the compile-time declaration of a call site. First, the callee method's signature along with a class or interface in which to search for a declaration matching the signature are computed. If in the source program the call simply consists of a method name, then the declaring class of the caller is searched. If the call consists of the name of a class type and a method name, then that class is searched. If the call consists of an expression and a method name, then the compile-time type of the expression is searched. If the call consists of the keyword `super` and a method name, then the superclass of the caller's declaring class is searched.

A method declaration is *applicable* to a call site if and only if the declaration and method specified at the call site have the same number of parameters and the type of each actual argument are assignment compatible with the declared type of the corresponding formal parameter. More than one method declaration may be applicable to an invocation. In this case, the most *specific* method is chosen. A method, `m`, declared in class `T` is more specific than a method, `m`, declared in `U` if and only if `T` is assignment compatible with `U` and each parameter type in `T` is assignment compatible with its corresponding parameter type in `U`. The most specific applicable method declaration for a call site is called the *compile-time declaration*. The second step searches the class produced by the first step to locate method declarations applicable to the call site.

If a call site's compile-time declaration is an instance method, then the call site is *dynamically bound*. Dynamically bound call sites require a run-time lookup to compute the method that is invoked. The target class instance on which a dynamically bound virtual invocation is performed in called the *receiver* of the invocation. If the method specified at the call site is a class method, then we say that the call site is *statically bound*. The method specified at a call site is the method that will be invoked at run time.

2.2.2 Method Invocation

The run-time invocation of a call to method m involves several steps. First, the target class instance of the invocation is computed, if the method invocation is dynamically bound. Then, the actual arguments of the invocation are evaluated. Next, the method to invoke is located. Statically bound invocations have no receiver and overriding is forbidden. So, the method invoked run-time is always the method to which the call site is statically bound. If the invocation is dynamically bound, a *dynamic method lookup* determines which method is invoked. The dynamic method lookup starts with the run-time type, S, of the receiver. S is searched for a method declaration that matches m. Two method declarations match if they have the name number and types of parameters and they have the same return type. If no method declared in S matches m, then the superclass of S is searched. This process is applied recursively up the class hierarchy until a match is found. Finally, program control is transferred to the method being invoked.[1]

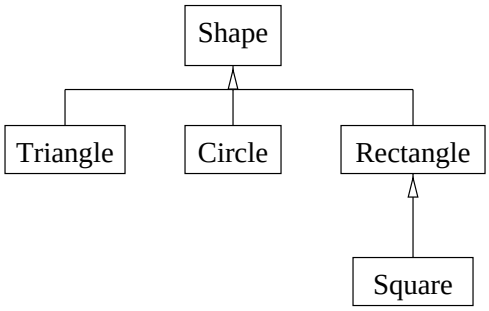
2.3 Type Analysis

Type analysis computes information about the types of various program entities. The analyses discussed here determine the possible types of the receiver of a dynamically bound call site and thus the set of possible methods that could be invoked. Type analysis creates a call graph for a program that is consulted when performing certain optimizations. Various aspects of the type analyses presented in this thesis will be demonstrated using the classes in Figure 2.1.

2.3.1 Class Hierarchy Analysis

Class hierarchy analysis [Dean et al. 1995; Fernández 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. Consider the getArea method in

[1] Note that most implementations of dynamic method lookup use caching techniques such as virtual tables to increase efficiency.




```
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); }
}
```

Figure 2.1: Example class hierarchy

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

Figure 2.2: Calling method area

Figure 2.2 which contains a call to the instance method area. Depending on the runtime type of the receiver of the method, the call could 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 possible be invoked is the area implementation in Rectangle because no subclass of Rectangle overrides area. This observation is key to class hierarchy analysis.

Class hierarchy analysis creates a program's call graph. A *call graph* is a directed graph that models the calling relationships among a program's methods. Each node in a call graph represents a method containing a set of call sites. The roots of the call graph correspond to the entry points of the program (e.g. the main method). Each edge represents one method (potentially) calling another. If method foo calls or may call method bar, there is a directed edge from the node representing foo to the node representing bar.

The call graph contains several important pieces of information. First, it demonstrates which methods may be invoked during the execution of the program. Such methods are referred to as being *live*. Only methods that have nodes in the call graph are interesting and need to be considered for further analysis. The call graph reveals which methods could potentially be invoked at a call site. If only one method can be invoked at a call site, then the site is said to be *monomorphic*. Otherwise, it is said to be *polymorphic*. Call sites that are monomorphic do not require a dynamic method lookup at run-time.

2.3.2 Rapid Type 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 2.2. 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 is considered 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.

Rapid type analysis traverses a program's call graph starting at its root methods. 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

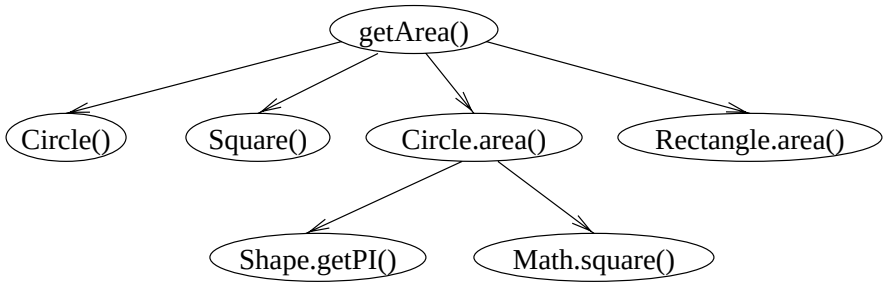


Figure 2.3: The call graph for getArea

methods are all live. The area method of Triangle is blocked on Triangle. The call graph compute by using rapid type analysis for getArea is given in Figure 2.3.

2.4 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 the compilation phase, then the call site can be statically bound.

The call to area 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 virtual invocations with two type tests and corresponding statically bound invocations (in the form of a call to a class method) as show in Figure 2.4. 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 virtual 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);
}
```

Figure 2.4: Customized call to area

2.5 Preexistence

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 reoptimized.

This situation is further exacerbated by the fact that optimized code may need to be reoptimized while it is executing. Consider the program in Figure 2.5a. 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 deoptimization of optimized code. When an optimization is invalidated, SELF recovers the original source code and

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

(a) Receiver does not preexist

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

(b) Receiver preexists

Figure 2.5: Preexistence of receiver objects

re-optimizes it taking the invalidating information into account. Maintaining the amount of information necessary to perform these kinds of optimizations requires a noticable 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 *preexistence* to eliminate the need for on-stack replacement. Consider a method foo containing a invocation of method bar with receiver object o. o is said to *preexist* if it is created before foo is called. The type of any preexistent 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 effect the method foo. 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 2.5b. In this method the receiver of the virtual method invocation 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.

An object may be proven to be preexistent using two techniques. Obviously, a method's arguments and receiver are created before the method is invoked. *Invariant argument analysis* uses this fact and traces the usage of the arguments throughout the method. If an argument is used as the receiver of a method invocation, that call may safely be optimized without worrying about on-stack replacement.

Immutable field analysis considers private instance fields whose values are assigned to only in constructors. Constructors are called before the object being initialized can be used. If the value of the field is used as a receiver of a method invocation and it is known that

the field has not be modified since its object's construction, then we can state the receiver object preexists.

2.6 Orthogonal Persistence

Large applications tend to have data that outlives a single program execution. Thus, the need for data management software arises. The object-oriented programming paradigm takes a data-centric view of software. It seems natural for object-oriented data to persist between program executions.

Orthogonal persistence [Atkinson et al. 1983; Atkinson and Morrison 1995] allows for automatic storage of program data by integrating a program's runtime environment with a stable data store. There are two driving principles behind persistence: *transparency* and *orthogonality*. Transparency states that programs that operate on persistent data cannot be distinguished from programs that operate on transient data. Transparency is especially important in that it allows software developed for transient data to be reused with persistent data. Orthogonality ensures that any data may persist regardless of its type. Orthogonality ensures that programmers are not required to identify persistent data at the time it is created. Most persistent systems implement persistence by reachability: if an object has been designated a *persistence root* or is referenced by another persistent object, it itself is persistent.

This thesis uses an orthogonally persistent Java virtual machine to analyse and optimize Java programs. Various kinds of analysis and optimization information are maintained as persistent Java objects. This information can be consulted at runtime to determine whether any changes to the type system have been made that invalidate the optimizations.

2.7 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] presents a constraint-based algorithm for interprocedural type inferencing that operates in $O(n^3)$ time where n is the size of the program. Ageson [Agesen 1994] presents a survey of various improvements to the $O(n^3)$ algorithm. Several the strategies discussed create copies of methods called "templates" whose type information is specialized with respect to the type of the parameters. Ageson also describes the "Cartesian Product Algorithm" [Agesen 1995] that creates a template for every receiver and argument tuple on a per-call site basis. 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] uses class hierarchy analysis and an intraprocedural algorithm in addition to a context-insensitive type propagation algorithm to optimize Modula-3 programs. Budimlic and Kennedy present interprocedural analyses and method inlining of Java programs [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; Fernández 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.

3 IMPLEMENTATION

3.1 The Java Virtual Machine

The Java virtual machine [Lindholm and Yellin 1996] is an abstract machine that executes programs specified by the binary class file format. The class file format and the Java virtual machine's instruction set, called *bytecodes*, support all of the operations necessary to execute programs written in the Java programming language. The Java virtual machine contains a dynamic allocation heap, method area, run-time constant pool, and certain per-thread data structures. The *heap* is a memory area, shared among all threads, in which objects and arrays are allocated. It is the responsibility of an automatic memory manager to deallocate the memory in the heap. The *method area* contains per-class data structures such as the run-time constant pool and the code for methods and constructors. The *run-time constant pool* holds constants representing program elements ranging from numeric literals to symbols representing fields and methods.

During execution, a *frame* is created whenever a method is invoked. Each frame has a list of local variables, an operand stack, and a reference to the run-time constant pool of the class containing the method being invoked. *Local variables* hold values accessed during method execution. Specifically, the *n* arguments of the method are stored in the first *n* local variables of the method's frame. For instance methods, the receiver object is always stored in the first local variable. The *operand stack* is used to hold partial results and for passing parameters to methods.

The Java virtual machine's instruction set operates primarily on values residing in local variables or on the operand stack. There are instructions to perform basic arithmetic

operations such as addition, multiplication, bitwise AND, and numerical comparison; object creation, operand stack manipulation, control transfer, method invocation, and thread synchronization.

Of particular interest to this work are the Java virtual machine's instructions for invoking methods. Each method invocation instruction has operands that index an entry in the constant pool describing the method being invoked. This method description includes the name, number and type of arguments, and the return type of the method. Prior to a method invocation the arguments to the method are pushed onto the operand stack. The Java virtual machine's method invocation mechanism pops the arguments off of the caller's stack and stores them into into the callee's first *n* local variables where *n* in the number of arguments to the method.

There are four instructions that invoke methods: invokestatic, invokeinterface, invokevirtual, and invokespecial. invokestatic invokes static methods and transfers program control to the method being invoked provided that it exists and is static. invokeinterface and invokevirtual invoke an instance method declared in an interface or class, respectively. Both of them perform a "dynamic dispatch" and select the method to be invoked as follows. A class C is searched for a method matching the one being invoked. Initially, C is the run-time class of the receiver object of the method invocation. If C contains a method with the same name, argument types, and return type as the argument to the invocation instruction, then program control is transferred to that method. Otherwise, the above step is repeated for the superclass of C until a matching method is found. invokespecial is used to invoke instance initialization methods (constructors)[1] and special methods, such as superclass and private methods. If the method being invoked is private or protected, invokespecial ensures that the class of the receiver object is the same or a superclass of the class containing the method being invoked. Neither invokespecial nor invokestatic performs a dynamic dispatch, thus they are said to be *non-virtual* method invocations.

[1]The Java programming language syntax new Foo() is compiled into an object allocation instruction, new Foo, and a invocation of the constructor method, invokespecial Foo.<init>.

3.2 Modeling Java Programs

We used the Bytecode-Level Optimizer and Analysis Tool [Nystrom 1998], BLOAT, to model Java class files. BLOAT contains a mechanism for reading class files from within a persistent store and modeling them as Java objects. BLOAT models classes, methods, and fields, resolves constants from the class file's constant pool, and represents a method's code as a series of instructions with optional arguments and labels that are the targets of jump instructions.

BLOAT also maintains a data structure for modeling the class hierarchy for the classes on which it operates. The class hierarchy represents both the inheritance relationships among classes and the implementation relationships between interfaces and classes.

Unlike traditional programming languages, whole-program analysis of a Java program is difficult. The Java programming language allows classes to be dynamically loaded at any time during execution. As such, performing static analyses on all of the classes required by a Java program is not always possible. Descriptions of the types of the objects, methods, and fields accessed by a Java class reside in the class's constant pool. BLOAT uses these constants to determine the names of classes that may be accessed by a Java program. Unfortunately, this analysis cannot account for classes referenced by native methods nor for classes loaded via the class reflection mechanism.

3.3 Call Graph Construction

The standard class library for the Java 2 platform consists of over 4600 classes. In order to handle so many classes in a practical manner, rapid type analysis is incorporated into the construction of the call graph. The call graph construction algorithm operates on a worklist of methods that initially contains the entry point of the Java program. In addition to the expected "who calls who" information, the call graph maintains a set of methods and a set of classes that are considered to be "live". The hybrid algorithm for call graph construction given in Figure 3.1 proceeds as follows. First, the call sites in each method are examined.

If the call site is non-virtual, then there is no dynamic dispatch and the callee method is made live and added to the worklist.

If the call site is virtual, the class hierarchy is consulted to determine the set of methods that could be invoked. The description of the virtual callee, m, not only contains its name and signature, but also contains the name of the class, C, the declared class of the receiver. The RTA algorithm examines C and all of its subclasses and looks for classes that override m. Any of these overriding methods could be invoked. It is at this point that rapid type analysis is applied. Each of the possible callees, m_i, is examined. If the class in which m_i is declared (or any of its subclasses in which m is not overridden) has been instantiated, then m_i is considered to be live and is added to the worklist. If neither the class in which m_i is declared nor any of its non-overriding subclasses has been instantiated, then m_i is "blocked" on each of those classes.

The rapid type analysis algorithm described in Section 2.3.2 relied on constructor invocations to determine which classes were live. Examining constructors in that manner is not necessary in our implementation. The Java Virtual Machine's new instruction instantiates an object of given class and thus makes that class live. When a class becomes live, any method that was blocked on that class is added to the worklist.

There are several Java-specific issues that must be dealt with during call graph construction. The Java virtual machine instantiates a number of classes internally. Consider a program that contains nothing but System.out.println("Hello World"). Since no class is instantiated in the program itself, the call graph would state that no method would be invoked by the call to println. Clearly, this is incorrect. To remedy this problem, our optimizer designates a number of classes to be *pre-live*.

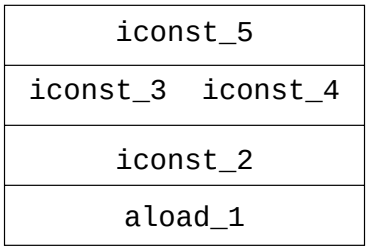
Classes may have static initializer methods that are invoked implicitly by the virtual machine the first time a class is referenced. Call graph construction considers a class's static initializer method to be live when the class is instantiated, one of its methods is invoked, or when one of its static fields is referenced.

```
input:
   A set of root methods, roots
output:
   The call graph, callGraph, representing the invocation relationship among methods.

do
   callGraph ← ∅
   liveClasses ← ∅
   worklist ← roots

   while (worklist ≠ ∅) do
      Remove a method, method, from the worklist
      for each instruction inst in method do
         if (inst instantiates a class) then
            type ← class being instantiated
            makeLive(type)
         else if (inst invokes a virtual method) then
            callee ← virtual method being called
            doVirtual(method, callee)
         else if (inst invokes a non – virtual method) then
            callee ← method being called
            callGraph(method) ← callGraph(method) ∪ callee
            worklist ← worklist ∪ {callee}
with
   procedure makeLive(type) begin
      liveClasses ← liveClasses ∪ type
      for each method blocked on type do
         worklist ← worklist ∪ type

   procedure doVirtual(caller, callee) begin
      for each method the callee may resolve to do
         isLive ← false
         for each possible receiver type, rType, of method do
            if (rType ∈ liveClasses) then
               isLive ← true
               worklist ← worklist ∪ method
               callGraph(method) ← callGraph(method) ∪ callee
               break
            if (not(live)) then
               for each possible receiver type, rType, of method do
                  block method on rType
```

Figure 3.1: Call graph construction using rapid type analysis

```
 0 aload_1
 1 iconst_2
 2 iload_2
 3 ifeq 10
 6 iconst_3
 7 goto 11
10 iconst_4
11 iconst_5
12 invokevirtual A.g(III)
15 return
```

(a) Compiled bytecode



(b) The instruction stack

Figure 3.2: Instruction stack for `a.g(2, (b?3:4), 5)`

3.4 Call Site Customization

Customization examines each virtual method invocation and uses information obtained from the call graph to determine which methods could be invoked. It then adds code to the caller that performs a "case switch" on the receiver object and replaces the virtual method call with a non-virtual method call, thus eliding the run-time overhead of the dynamic dispatch.

Customization must locate the instructions that pushes the receiver object onto the operand stack. Finding these instructions is more difficult that it may appear. Because of language constructs such as the conditional operator (?:), there may be arbitrary control flow between the invocation instruction and the instructions that push its receiver on the stack. Thus, the contents of the stack must be simulated as the method is examined. Each element in the simulated stack is a set of instructions that are responsible for pushing values at that stack height. Such a stack is demonstrated in Figure 3.2. For an invocation of a method with *n* parameters, the instructions that push the receiver onto the stack are located depth *n* in the instruction stack.

Maintaining the stack of instructions is essential to implementing preexistence. Recall that only call sites whose receiver objects preexist may safely be inlined and, thus, are worthwhile to customize. Before a call site is customized the instructions that push the receiver on the operand stack object are examined. If those instructions load preexistent

local variables or create objects, then the receiver preexists and may be safely customized. However, a receiver that results from a load of an object's field or a method call do not preexists and cannot safely be customized. While preexistence restricts the number of call sites that may be customized, it does have a benefit. If all of the instructions that push the receiver on the stack are object creation instructions, then the possible run-time types of the receiver object are known. These types are used to reduce the set of methods that may be invoked at the call site. Thus, as a byproduct of prexistence the implementation of rapid type analysis performs a limited dataflow analysis on the type of the receiver object.

The following steps are taken to customize a virtual invocation of a method m and are illustrated for the area method in Figure 3.3. The instructions that push the receiver of the call are located. Instructions are added to the caller that duplicate the receiver object and store it in a local variable. Then at the call site, the receiver object is loaded from the local variable. If the call site is polymorphic, code is added that tests the type of the receiver. If the type of the receiver matches a class C, a non-virtual invocation of the implementation of the m method in class C is performed using the invokespecial instruction. Note that the order in which the type checks occur is important. The instanceof instruction does not test type equality – any subtype of the request type is also an instance of the desired type. Thus, the more "refined" types must be tested for first.

The implementation of customization differs from the description given in Section 2.4 in that no static version of the virtual method is generated. Making a static versions of methods not only increases the size of the optimized class files, but also has a negative impact on the instruction and data caches at runtime. However, in order to use the invoke-special instruction, the virtual machine was modified so that any method, not just private and protected methods, could be invoked non-virtually.

3.5 Method Inlining

The final phase of the interprocedural optimizations is to inline calls to non-virtual methods. Calling non-virtual methods involves no dynamic dispatch, so it is known what

```
...
aload_2
invokevirtual Shape.area()F
fstore_3
fload_3
freturn
```

(a) A virtual call

```
        ...
        aload_2
        dup
        astore_4
        aload_4
        instanceof Circle
        ifeq NEXT1
        invokespecial Circle.area()F
        goto END
NEXT1:  aload_4
        instanceof Rectangle
        ifeq DEFAULT
        invokespecial Rectangle.area()F
        goto END
DEFAULT: invokevirtual Shape.area()F
        goto END
END:    fstore_3
        fload_3
        freturn
```

(b) Customized call site

Figure 3.3: Customizing a virtual call site

code will be executed by the call. Provided that certain conditions are met, this code may be copied into the caller method and the overhead of calling the method is removed.

The process of inlining is relatively straightforward. Each non-virtual call site is examined. If it meets the criteria outlined below, then call is inlined. Inlining involves copying the callee's code into the caller method and changing the names of local variables and labels in the callee so that they do not conflict with those of the caller. First, instructions are added to store the arguments from the stack into local variables. Then the callee's code is copied into the caller. Local variables and labels in the callee method are mapped to non-conflicting local variables and labels in the caller method. An example of an inlined method call is given in Figure 3.4.

There are several situations in which a call to a non-virtual method is not inlined. Recursive calls and calls to native methods are not inlined. Because invoking a synchronized method involves obtaining a lock, synchronized methods are not inlined. Additionally, some methods that may catch exceptions cannot be inlined. When an exception is caught, the operand stack is cleared [Lindholm and Yellin 1996]. For an inlined method, clearing

the stack effects the execution of the caller method as well. Thus, when the inlined method "returns", the stack state is not as expected. The only circumstance in which a call to a method that may catch an exception may safely be inlined occurs the only values on the operand stack at the time of the call are the callee's arguments.

It is also possible to inline certain constructor invocations. The first call made by a constructor is a call to its superclass's constructor. There is no dynamic dispatch that occurs here and no object is created. Thus, the code of the superclass's constructor can be inlined.

Inlining methods that access non-public data invalidates the encapsulation assertions made about Java methods during compilation. Additionally, other optimizations may result in code that does not verify. We argue that once a class has become resident in a persistent store, we need not worry about encapsulation and verification. As classes are loaded into the system, they are verified to be well-behaved. As such, our Java virtual machine does not perform data access checks.

3.6 Intraprocedural Optimizations

BLOAT performs several intraprocedural optimizations on Java methods [Nystrom 1998]. For each method a control flow graph in static single assignment form is constructed. Optimizations such as expression propagation, dead code elimination, and partial redundancy elimination [Chow et al. 1997] of access expressions are performed on the control flow graph. After the control flow graph has been destructed, register allocation with graph coloring is applied to make efficient use of the Java virtual machine's local variables. Special analyses are performed to make better use of the Java virtual machine's stack manipulation instructions [VanDrunen 2000]. Finally, several peephole optimizations are applied to the generated instructions. Invoking methods restricts some optimizations because intraprocedural analyses do not analyze the behavior of the callee. Inlining gives BLOAT a larger context upon which to perform its intraprocedural optimizations.

3.7 Persistence-Enabled Optimization

The Java Language Specification [Gosling et al. 1996] requires that changes made to Java classes are *binary compatible* with preexisting class binaries. However our optimizations break Java's data encapsulation model by allowing caller methods to access the private data of inlined methods. Thus, our optimizations must be performed on classes in a safe environment in which the restrictions of 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 orthogonally persistent Java virtual machines maintain a representation of classes, as well as instantiated objects, in the persistent store. Such a store would give a good approximation of a closed-world scenario in which a Java program may be run. Additionally, the classes in the persistent store are verified to be binary compatible upon their entry to the virtual machine. 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, then 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. Thus, we can safely perform our optimizations on classes residing within the persistent store.

3.7.1 Deoptimization

Class hierarchy analysis and call site customization make certain assumptions about the classes present in the virtual machine. For instance, it is assumed that certain methods are not overridden by subclasses. However, it is possible that native methods or class reflection may bring classes into the system that break the assumptions. Thus, optimized code may need to be deoptimized 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 preexists 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 loads the EquilateralTriangle class, a subclass of Triangle that overrides the area method, using Java's reflection mechanism. During customization we assumed that no subclass of Triangle overrode the area method. However, the introduction of the EquilateralTriangel 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. Thus, we must deoptimize foo at runtime by undoing the effects of customization. In an attempt to make deoptimization 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 deoptimized. The optimizer maintains a series of dependencies [Chambers et al. 1995] among methods resulting from call site customization. Note that if our analysis can precisely determine the type(s) of a receiver object (e.g., the instructions that push the receiver onto the stack are all object creation instructions), then no dependence is necessary. The dependencies are represented as Java objects and reside in the persistent store.

The persistent Java virtual machine was modified to communicate with the optimizer at runtime to determine when methods should be deoptimized. 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 deoptimizes the appropriate methods. To account for any degradation in performance that deoptimization may produce, it may be desirable to optimize a Java program multiple times during its lifetime. Subsequent 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 deoptimization to ensure correct program behavior when classes are introduced into the system at runtime. Thus, persistence enables us to safely perform our interprocedural optimizations on Java programs.

```
          ...
          aload_2
          dup
          astore_4
          aload_4
          instanceof Circle
          ifeq NEXT1
          astore_5     // Store parameter
          aload_5
          invokevirtual Shape.getPI()F
          aload_5
          getfield Circle.r F
          invokestatic Shape.square(F)F
          fmul         // Return value on stack
          goto END
NEXT1:    aload_4
          instanceof Rectangle
          ifeq DEFAULT
          astore_6
          aload_6
          getfield Rectangle.s1 F
          aload_6
          getfield Rectangle.s2 F
          fmul
          goto END
DEFAULT:  invokevirtual Shape.area()F
          goto END
END:      fstore_3
          fload_3
          freturn
```

Figure 3.4: Inlining customized code

4 EXPERIMENTS

To evaluate the impact of interprocedural optimizations on Java programs, we optimized several Java benchmark applications and compared their performance using static and dynamic performance metrics.

4.1 Platform

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

4.2 Benchmarks

We used eleven benchmarks programs as described in Table 4.1 to measure the impact of interprocedural optimizations. Several of the benchmarks were taken from the SpecJVM [SPEC 1998] suite of benchmarks. Table 4.2 gives some static statistics about the benchmarks: the number of live classes and methods, the number of virtual call sites, the percentage of those calls sites that preexist, and the percentage of preexistent 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 preexist. Note also that nearly all (89.8–95.7%) of preexistent

Table 4.1: 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

Table 4.2: 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

call sites are monomorphic or duomorphic. Thus, from a static point of view, extensive customization of polymorphic call sites seems unnecessary.

Each benchmark was optimized in five configurations: no optimizations, (nop), only intraprocedural optimizations (intra), call site customization (cust), inlining of non-virtual calls (inline), and intraprocedural optimizations on top of inlining (both). Through analysis of empirical data, several conditions on the interprocedural 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.

4.3 Execution environments

The benchmarks were executed in three different execution environments: the Sun Java 2 SDK Solaris™ Production Release Virtual Machine with and without a Just-In-Time compiler (labelled JIT and noJIT, respectively), and the Toba bytecode-to-C compiler version 1.1 [Proebsting et al. 1997; Toba 1998].

4.4 Metrics

Several dynamic metrics were used to measure the impact of our optimizations. We measure the number of cycles, instructions executed, data cache read misses, and instruction cache misses for each benchmark using software [Enbody 1998] that allows user-level access to the UltraSPARC's execution counters. Each benchmark was run twice inside a single activation of the execution environment (JIT, noJIT, Toba). The first iteration primes the execution environment: class files are loaded, bytecodes are JIT compiled, and the caches are warmed. The measurements reported here were taken during the second iteration of the benchmark.

The instruction cache on the UltraSPARC is physically addressed and, therefore, program behavior with respect to the instruction and data caches may vary noticeably across executions. To account for this deviation the dynamic measurements reported are the average over 3 runs of the benchmark and 90% confidence intervals are shown in the graphs.

4.5 Results

The results for each benchmark are summarized in Tables 4.3-4.12. When reporting the counts of the executed bytecode raw data is reported for nop only. We give the totals for the other optimization levels as a portion of nop's total. The count of individual bytecodes are reported as a portion of the total number for the given optimization level. Similarly, we give the raw data for the hardware execution counters for nop; while the other data are reported as a ratio relative to nop.

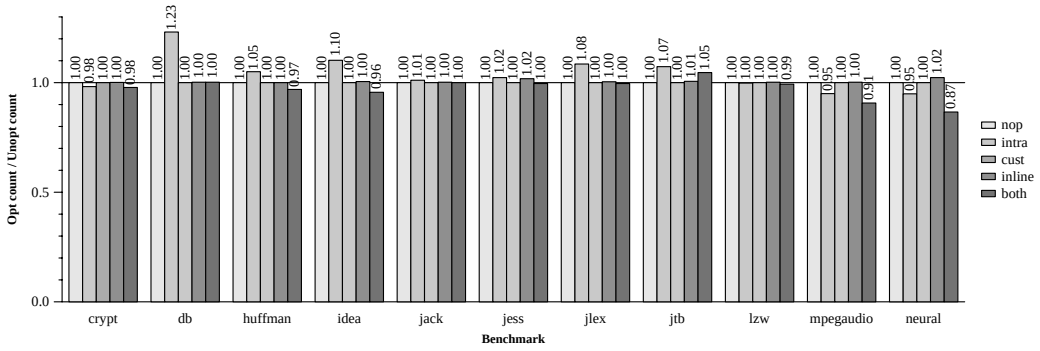


Figure 4.1: Total bytecodes executed

4.5.1 Bytecode counts

Examining the number of bytecodes executed provides insight into the effectiveness of our interprocedural optimizations. Figures 4.1, 4.2, and 4.3 summarize bytecode counts for the five optimization levels: nop, intra, cust, inline, and both. As Figure 4.1 demonstrates the interprocedural optimizations, cust and inline, do not have a signifcant effect on the total number of bytecodes executed. However, combining interprocedural and intraprocedural optimizations (both) results in up to 8% fewer bytecodes being executed than with the intraprocedural 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 4.2 reports the number of invokespecial, invokevirtual[1], 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 inlinable methods.

Recall that when a method call is inlined, the callee's parameters must be popped from the operand stack into local variables. This results in a noticeable increase in the number of

[1]There were a negligable number of invokeinterface instructions executed.

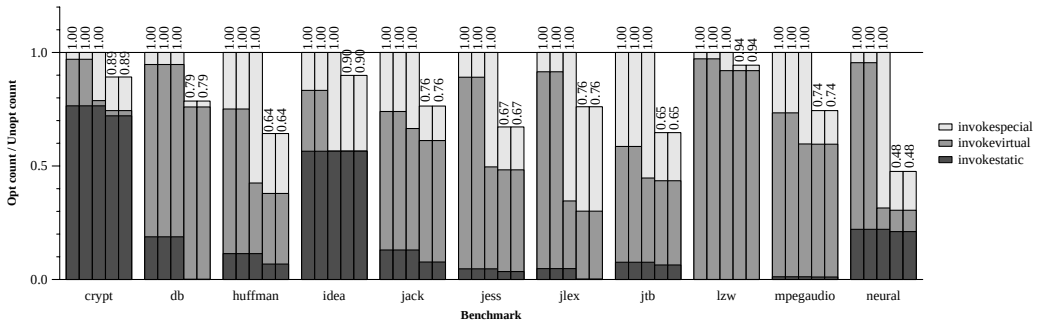


Figure 4.2: Bytecode counts of method invocations

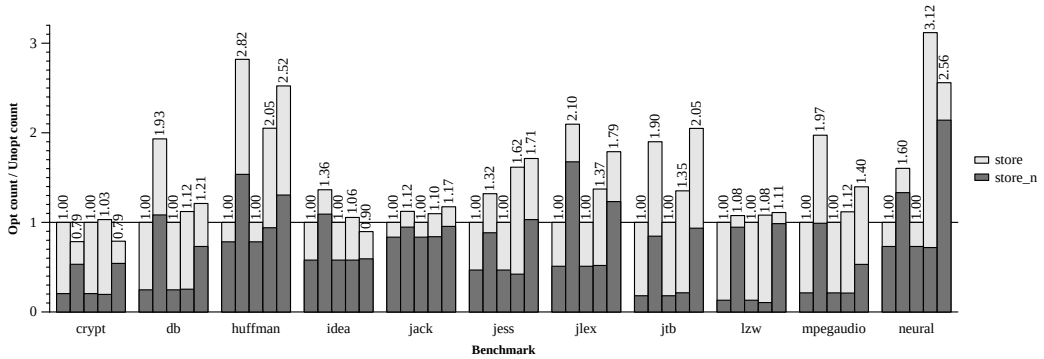


Figure 4.3: Store bytecodes

store instructions as shown in Figure 4.3. While there is a performance penality for executing these additional stores, we argue that this penalty is overshadowed by the performance gains brought about by inlining.

4.5.2 noJIT

We saw our greatest performance improvement when executing optimized code under the bytecode interpreter (noJIT). As Figure 4.4 demonstrates, all benchmarks show some improvement when methods are inlined. Our optimizations cause a 3–23% decrease in the number of machine instructions executed. In general the decrease in the number of cycles is not as drastic. For several benchmarks, our optimizations cause an increase in the number of instruction fetch stalls and data read misses.

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 intraprocedural optimizations to increase performance further.

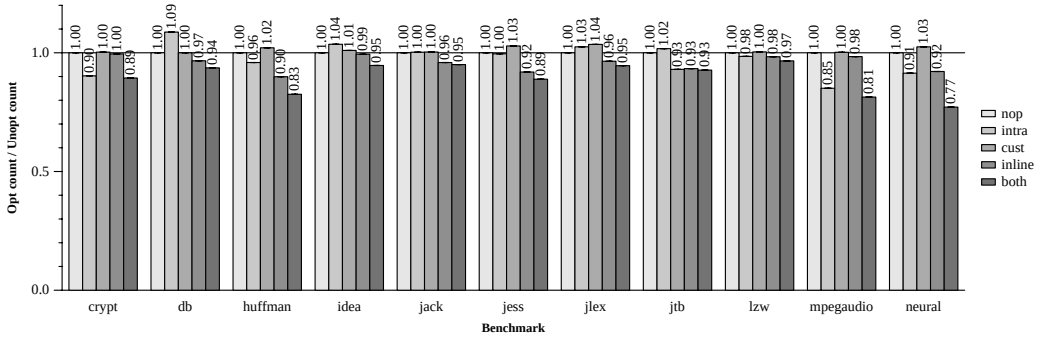
4.5.3 JIT

A Just-In-Time (JIT) compiler compiles Java bytecode instructions into native machine instructions which are then executed on bare hardware. The JIT we used to measure our benchmarks performs a number of optimizations including call site inlining [Detlefs and Agesen 1999] on the code it compiles. Provably monomorphic call sites are inlined directly. The inlined code for call sites that are almost monomorphic are guarded by a run-time "method guard" that compares the method about to be executed to the method that was inlined.

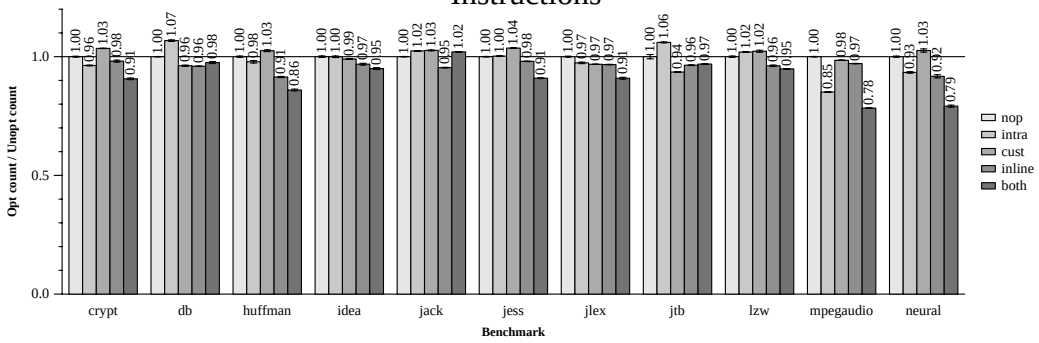
It is not surprising, then, that our optimizations conflict with those performed by the JIT compiler. Our method inlining may increase the size of a method beyond a point where the JIT compiler can generate efficient code. As Figure 4.5 demonstrates[2] the method inlining performed by our optimizations (inline) offers little advantage over the unoptimized code. However, performing intraprocedural optimizations over the inlined code (both) does reduce the number of instructions executed in most cases.

Inlining degrades the number of cycles executed by most benchmarks from 2–26%. This is primarily due to an increase in the number of misses in the data and instruction caces. The intraprocedural optimizations fare much better than the interprocedural optimizations in the JIT environment.

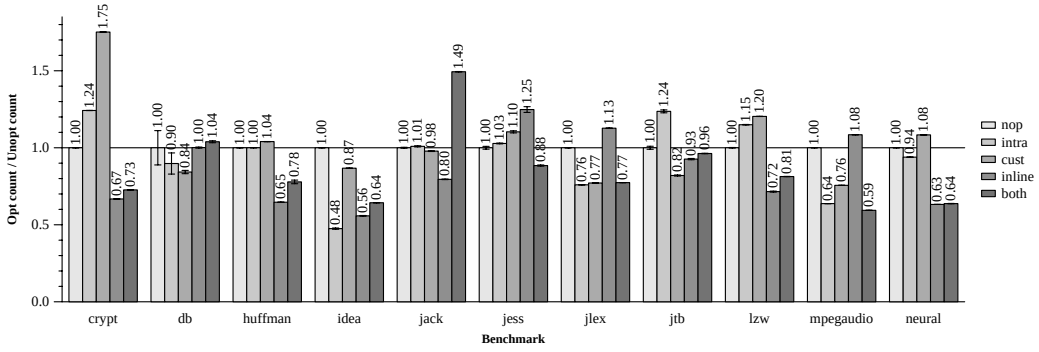
[2]Two benchmarks, jess and jtb, when optimized with the intraprocedural (intra) optimizations cause the JIT to generate code that results in an address alignment error at run-time. As this error does not occur with the interpreter, we believe this behavior to stem from a bug in the JIT.



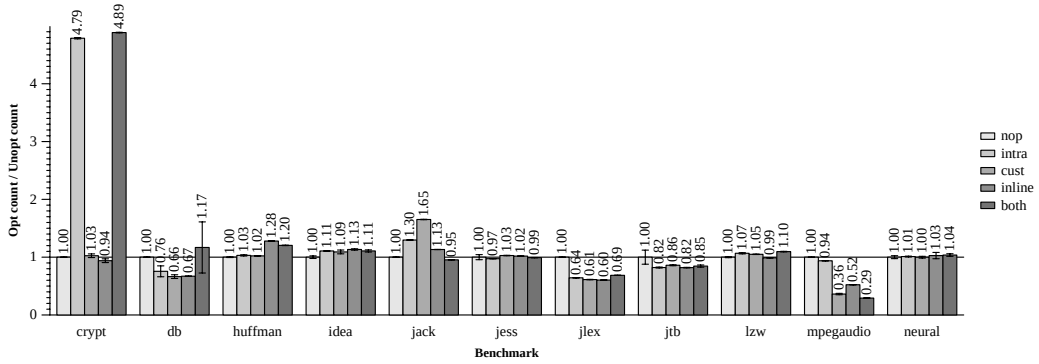
Instructions



Cycles

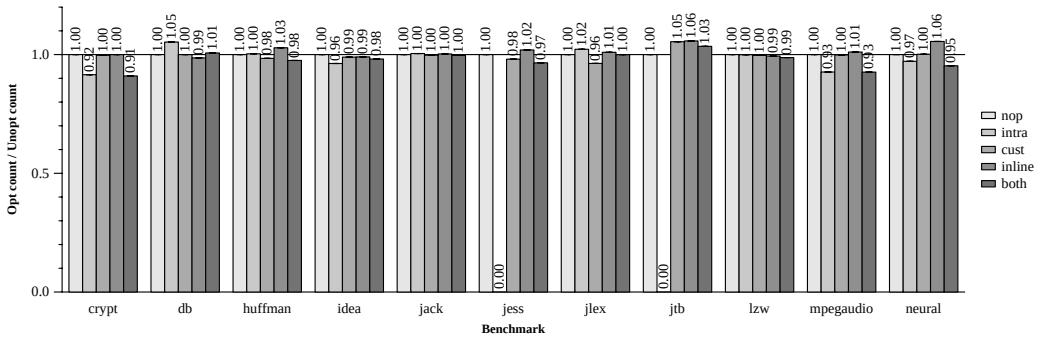


Data Cache Misses

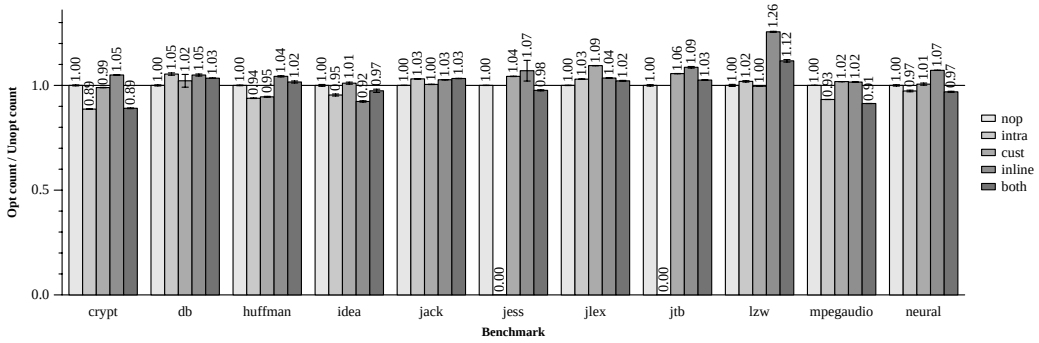


Instruction Cache Misses

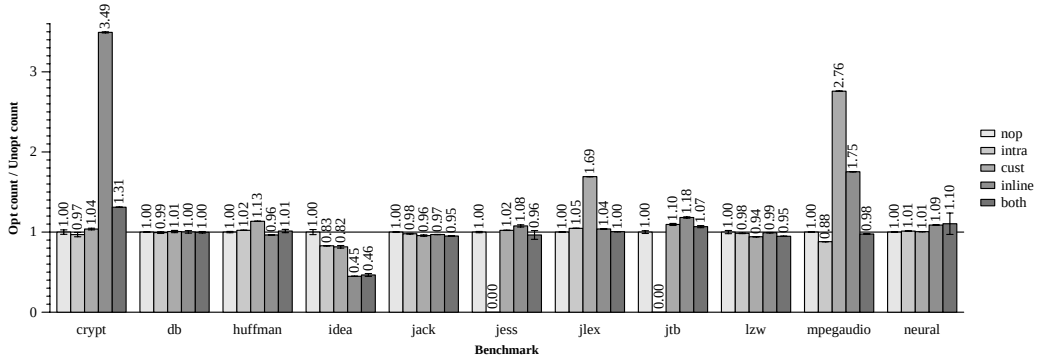
Figure 4.4: Execution time for noJIT



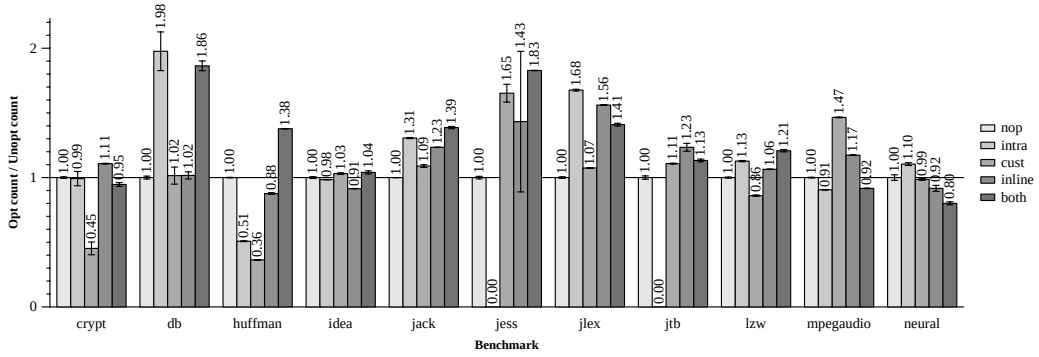
Instructions



Cycles



Data Cache Misses



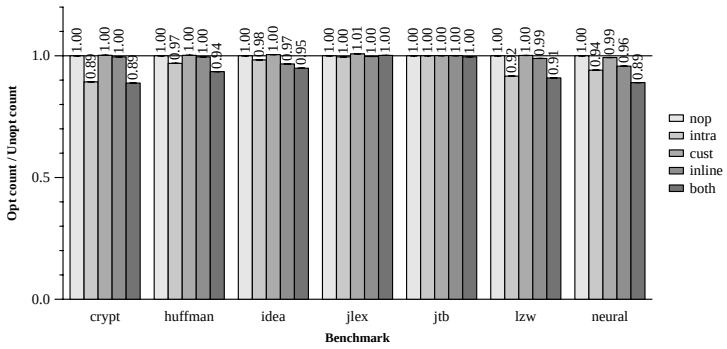
Instruction Cache Misses

Figure 4.5: Execution time for JIT

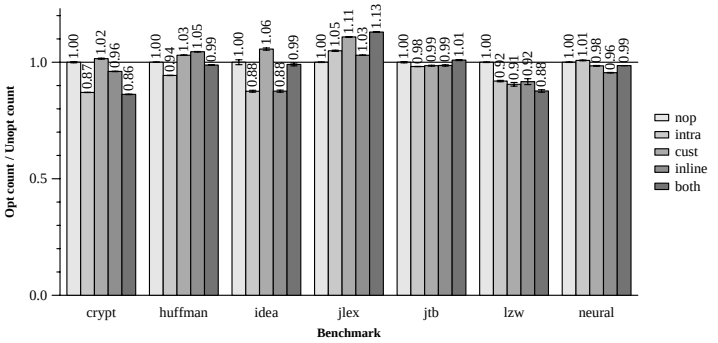
4.5.4 Toba

Toba applies an optimizing C compiler to a Java program. We used Sun's WorkShop C Compiler version 5.0 which performs local and global optimizations such as induction variable elimination, algebraic simplification, copy propagation, constant propagation, loop-invariant optimization, register allocation, basic block merging, tail recursion elimination, dead code elimination, tail call elimination and complex expression expansion. We hypothesize that the C compiler would take advantage of the larger code context provided by our interprocedural optimizations. The version of Toba that we used runs with Java 1.1 and does not fully support the entire Java class library, in particular the abstract windowing toolkit. As a result, the SpecJVM benchmarks could not be run under Toba.

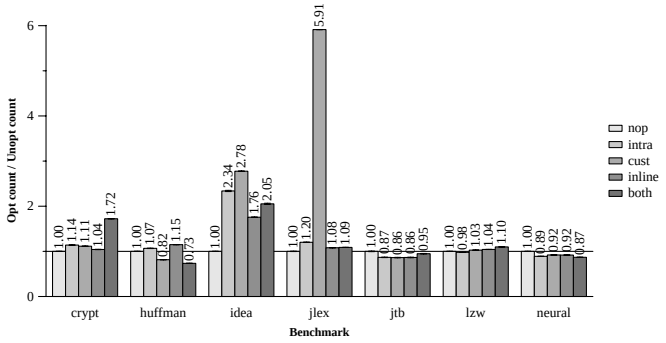
The results for toba are difficult to characterize. The number of instructions and cycles are summarized in Figure 4.6. For some benchmarks our optimizations had no impact on the number of instructions performed. In others we saw a decrease of 5–11% in the number of instructions executed. Once again the decrease in the number of instruction did not always cause a corresponding decrease in the number of cycles due to caching behavior.



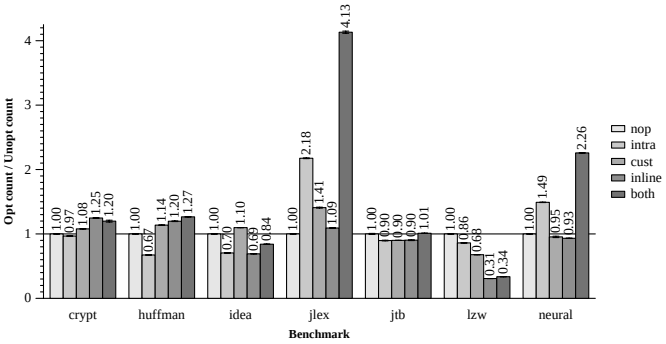
Instructions



Cycles



Data Cache Misses



Instruction Cache Misses

Figure 4.6: Execution time for Toba

Table 4.3: Results for crypt

Platform	Metric	nop	intra	cust	inline	both
Bytecodes	TOTAL	260853720	0.98	1.00	1.00	0.98
	invokevirtual	0.18	0.18	0.02	0.02	0.02
	invokespecial	0.03	0.03	0.19	0.13	0.13
	invokestatic	0.68	0.69	0.68	0.64	0.65
	dup	0.02	4.67	0.02	0.02	4.76
	if	0.49	0.50	0.49	0.49	0.50
	ifcmp	0.75	0.70	0.75	0.75	0.70
	load	15.25	4.15	15.25	15.62	4.14
	load*n*	10.34	20.72	10.34	9.95	20.56
	store	4.68	1.51	4.68	4.91	1.49
	store*n*	1.21	3.19	1.21	1.15	3.26
JIT	Cycles	328642884	0.89	0.99	1.05	0.89
	SPARC instructions	337054347	0.92	1.00	1.00	0.91
	Instruction fetch stalls	1819446	0.99	0.45	1.11	0.95
	Data read misses	1345925	0.97	1.04	3.49	1.31
noJIT	Cycles	5926082142	0.96	1.03	0.98	0.91
	SPARC instructions	4601475214	0.90	1.00	1.00	0.89
	Instruction fetch stalls	7430311	4.79	1.03	0.94	4.89
	Data read misses	27265184	1.24	1.75	0.67	0.73
Toba	Cycles	465352567	0.87	1.02	0.96	0.86
	SPARC instructions	471584726	0.89	1.00	1.00	0.89
	Instruction fetch stalls	8623059	0.97	1.08	1.25	1.20
	Data read misses	1169225	1.14	1.11	1.04	1.72

Table 4.4: Results for db

Platform	Metric	nop	intra	cust	inline	both
Bytecodes	TOTAL	3747617527	1.23	1.00	1.00	1.00
	invokevirtual	2.44	1.98	2.44	2.44	2.44
	invokespecial	0.17	0.14	0.17	0.08	0.08
	invokestatic	0.60	0.49	0.60	0.00	0.00
	dup	0.53	2.55	0.53	0.53	3.49
	if	3.49	2.83	3.49	3.48	3.48
	ifcmp	4.54	3.69	4.54	4.53	4.53
	load	17.18	11.69	17.18	19.29	10.43
	load*n*	23.32	26.87	23.32	21.13	27.62
	store	8.07	7.39	8.07	9.27	5.13
	store*n*	2.64	9.42	2.64	2.72	7.81
JIT	Cycles	16589918832	1.05	1.02	1.05	1.03
	SPARC instructions	8201809757	1.05	1.00	0.99	1.01
	Instruction fetch stalls	30612773	1.98	1.02	1.02	1.86
	Data read misses	529016289	0.99	1.01	1.00	1.00
noJIT	Cycles	136721160807	1.07	0.96	0.96	0.98
	SPARC instructions	93443093886	1.09	1.00	0.97	0.94
	Instruction fetch stalls	3668078414	0.76	0.66	0.67	1.17
	Data read misses	2025594158	0.90	0.84	1.00	1.04

Table 4.5: Results for huffman

Platform	Metric	nop	intra	cust	inline	both
Bytecodes	TOTAL	28209276	1.05	1.00	1.00	0.97
	invokevirtual	4.35	4.15	2.12	2.12	2.19
	invokespecial	1.70	1.62	3.93	1.80	1.86
	invokestatic	0.78	0.74	0.78	0.47	0.48
	dup	1.50	5.38	1.50	1.50	6.37
	if	3.28	3.14	3.28	3.28	3.40
	ifcmp	5.98	5.68	5.98	5.98	6.16
	load	4.93	3.39	4.93	10.27	4.50
	load*n*	30.33	34.77	30.33	24.98	33.06
	store	0.51	2.86	0.51	2.60	2.94
	store*n*	1.83	3.42	1.83	2.20	3.15
JIT	Cycles	108113898	0.94	0.95	1.04	1.02
	SPARC instructions	71083483	1.00	0.98	1.03	0.98
	Instruction fetch stalls	8296524	0.51	0.36	0.88	1.38
	Data read misses	1654630	1.02	1.13	0.96	1.01
noJIT	Cycles	1259474121	0.98	1.03	0.91	0.86
	SPARC instructions	928308022	0.96	1.02	0.90	0.83
	Instruction fetch stalls	39012174	1.03	1.02	1.28	1.20
	Data read misses	17953533	1.00	1.04	0.65	0.78
Toba	Cycles	335528931	0.94	1.03	1.05	0.99
	SPARC instructions	182798226	0.97	1.00	1.00	0.94
	Instruction fetch stalls	32595924	0.67	1.14	1.20	1.27
	Data read misses	2049671	1.07	0.82	1.15	0.73

Table 4.6: Results for idea

Platform	Metric	nop	intra	cust	inline	both
Bytecodes	TOTAL	16389364	1.10	1.00	1.00	0.96
	invokevirtual	0.63	0.57	0.00	0.00	0.00
	invokespecial	0.40	0.36	1.03	0.78	0.82
	invokestatic	1.34	1.21	1.34	1.33	1.40
	dup	0.32	3.04	0.32	0.32	2.60
	if	2.84	2.61	2.84	2.82	3.01
	ifcmp	3.16	2.76	3.16	3.14	3.22
	load	15.66	8.65	15.66	17.36	9.60
	load*n*	23.13	27.46	23.13	21.25	26.01
	store	5.11	3.00	5.11	5.76	3.87
	store*n*	7.04	12.04	7.04	7.01	7.54
JIT	Cycles	83054534	0.95	1.01	0.92	0.97
	SPARC instructions	36752141	0.96	0.99	0.99	0.98
	Instruction fetch stalls	18531557	0.98	1.03	0.91	1.04
	Data read misses	696192	0.83	0.82	0.45	0.46
noJIT	Cycles	497895342	1.00	0.99	0.97	0.95
	SPARC instructions	328071065	1.04	1.01	0.99	0.95
	Instruction fetch stalls	32445160	1.11	1.09	1.13	1.11
	Data read misses	5402604	0.48	0.87	0.56	0.64
Toba	Cycles	39381466	0.88	1.06	0.88	0.99
	SPARC instructions	19982196	0.98	1.00	0.97	0.95
	Instruction fetch stalls	8874997	0.70	1.10	0.69	0.84
	Data read misses	57096	2.34	2.78	1.76	2.05

Table 4.7: Results for jack

Platform	Metric	nop	intra	cust	inline	both
Bytecodes	TOTAL	2117399842	1.01	1.00	1.00	1.00
	invokevirtual	1.66	1.64	1.46	1.46	1.46
	invokespecial	0.71	0.70	0.91	0.41	0.42
	invokestatic	0.36	0.35	0.36	0.21	0.21
	dup	1.09	1.69	1.09	1.09	1.72
	if	13.31	13.23	13.31	13.28	13.38
	ifcmp	2.21	2.11	2.21	2.20	2.14
	load	3.26	2.93	3.26	4.02	3.42
	load*n*	37.10	37.31	37.10	36.26	36.88
	store	1.43	1.50	1.43	2.22	1.89
	store*n*	7.20	8.08	7.20	7.23	8.24
JIT	Cycles	5341660760	1.03	1.00	1.03	1.03
	SPARC instructions	3785866048	1.00	1.00	1.00	1.00
	Instruction fetch stalls	300976350	1.31	1.09	1.23	1.39
	Data read misses	91920188	0.98	0.96	0.97	0.95
noJIT	Cycles	68265734731	1.02	1.03	0.95	1.02
	SPARC instructions	47436823706	1.00	1.00	0.96	0.95
	Instruction fetch stalls	1471555600	1.30	1.65	1.13	0.95
	Data read misses	1081027154	1.01	0.98	0.80	1.49

Table 4.8: Results for jess

Platform	Metric	nop	intra	cust	inline	both
Bytecodes	TOTAL	1870816906	1.02	1.00	1.02	1.00
	invokevirtual	5.31	5.19	2.83	2.78	2.84
	invokespecial	0.68	0.67	3.17	1.16	1.19
	invokestatic	0.30	0.29	0.30	0.21	0.22
	dup	0.53	3.06	0.53	0.52	3.28
	if	3.80	4.09	3.80	3.73	4.20
	ifcmp	6.67	6.15	6.67	6.56	6.31
	load	7.86	4.93	7.86	14.95	7.87
	load*n*	29.60	29.83	29.60	21.86	26.46
	store	3.21	2.57	3.21	7.08	4.13
	store*n*	2.83	5.21	2.83	2.50	6.25
JIT	Cycles	5794754361	0.00	1.04	1.07	0.98
	SPARC instructions	3778754420	0.00	0.98	1.02	0.97
	Instruction fetch stalls	122195315	0.00	1.65	1.43	1.83
	Data read misses	113649546	0.00	1.02	1.08	0.96
noJIT	Cycles	72984475270	1.00	1.04	0.98	0.91
	SPARC instructions	55609627990	1.00	1.03	0.92	0.89
	Instruction fetch stalls	1572800729	0.97	1.03	1.02	0.99
	Data read misses	920519682	1.03	1.10	1.25	0.88

Table 4.9: Results for jlex

Platform	Metric	nop	intra	cust	inline	both
Bytecodes	TOTAL	68770643	1.08	1.00	1.00	1.00
	invokevirtual	4.81	4.44	1.66	1.65	1.66
	invokespecial	0.47	0.44	3.63	2.54	2.56
	invokestatic	0.26	0.24	0.26	0.01	0.01
	dup	1.18	4.99	1.18	1.18	6.52
	if	2.46	2.63	2.46	2.45	3.09
	ifcmp	8.29	7.28	8.29	8.25	7.69
	load	8.44	4.87	8.44	10.61	6.49
	load*n*	29.65	31.33	29.65	27.32	28.37
	store	2.41	1.90	2.41	4.17	2.74
	store*n*	2.50	7.58	2.50	2.53	6.07
JIT	Cycles	246734828	1.03	1.09	1.04	1.02
	SPARC instructions	179993623	1.02	0.96	1.01	1.00
	Instruction fetch stalls	5347549	1.68	1.07	1.56	1.41
	Data read misses	4002602	1.05	1.69	1.04	1.00
noJIT	Cycles	2773572957	0.97	0.97	0.97	0.91
	SPARC instructions	2018604126	1.03	1.04	0.96	0.95
	Instruction fetch stalls	77551372	0.64	0.61	0.60	0.69
	Data read misses	36500397	0.76	0.77	1.13	0.77
Toba	Cycles	814004370	1.05	1.11	1.03	1.13
	SPARC instructions	569819698	1.00	1.01	1.00	1.00
	Instruction fetch stalls	19797472	2.18	1.41	1.09	4.13
	Data read misses	1154929	1.20	5.91	1.08	1.09

Table 4.10: Results for lzw

Platform	Metric	nop	intra	cust	inline	both
Bytecodes	TOTAL	28239091	1.00	1.00	1.00	0.99
	invokevirtual	6.84	6.85	6.47	6.46	6.52
	invokespecial	0.19	0.19	0.56	0.17	0.17
	invokestatic	0.00	0.00	0.00	0.00	0.00
	dup	0.11	1.29	0.11	0.11	1.92
	if	5.22	6.21	5.22	5.21	6.24
	ifcmp	2.35	1.37	2.35	2.34	1.38
	load	10.94	1.27	10.94	12.04	0.76
	load*n*	19.81	28.70	19.81	18.67	28.40
	store	5.89	0.89	5.89	6.60	0.85
	store*n*	0.88	6.42	0.88	0.71	6.73
JIT	Cycles	131940268	1.02	1.00	1.26	1.12
	SPARC instructions	64727474	1.00	1.00	0.99	0.99
	Instruction fetch stalls	19722185	1.13	0.86	1.06	1.21
	Data read misses	2895969	0.98	0.94	0.99	0.95
noJIT	Cycles	1230814848	1.02	1.02	0.96	0.95
	SPARC instructions	865390494	0.98	1.00	0.98	0.97
	Instruction fetch stalls	55818107	1.07	1.05	0.99	1.10
	Data read misses	20331187	1.15	1.20	0.72	0.81
Toba	Cycles	146124730	0.92	0.91	0.92	0.88
	SPARC instructions	67600608	0.92	1.00	0.99	0.91
	Instruction fetch stalls	20392287	0.86	0.68	0.31	0.34
	Data read misses	2268787	0.98	1.03	1.04	1.10

Table 4.11: Results for mpegaudio

Platform	Metric	nop	intra	cust	inline	both
Bytecodes	TOTAL	11490419455	0.95	1.00	1.00	0.91
	invokevirtual	0.69	0.73	0.56	0.56	0.62
	invokespecial	0.25	0.27	0.38	0.14	0.16
	invokestatic	0.01	0.01	0.01	0.01	0.01
	dup	1.07	6.41	1.07	1.07	6.87
	if	1.26	1.32	1.26	1.25	1.39
	ifcmp	2.05	2.10	2.05	2.05	2.20
	load	13.76	9.21	13.76	14.82	10.18
	load*n*	14.07	15.25	14.07	12.94	13.31
	store	2.07	2.72	2.07	2.38	2.51
	store*n*	0.56	2.74	0.56	0.55	1.54
JIT	Cycles	14451934821	0.93	1.02	1.02	0.91
	SPARC instructions	12796410118	0.93	1.00	1.01	0.93
	Instruction fetch stalls	95426652	0.91	1.47	1.17	0.92
	Data read misses	106884266	0.88	2.76	1.75	0.98
noJIT	Cycles	302332129721	0.85	0.98	0.97	0.78
	SPARC instructions	220150524918	0.85	1.00	0.98	0.81
	Instruction fetch stalls	10502891108	0.94	0.36	0.52	0.29
	Data read misses	2368059043	0.64	0.76	1.08	0.59

Table 4.12: Results for neural

Platform	Metric	nop	intra	cust	inline	both
Bytecodes	TOTAL	17493501	0.95	1.00	1.02	0.87
	invokevirtual	2.46	2.59	0.31	0.31	0.36
	invokespecial	0.15	0.16	2.29	0.56	0.66
	invokestatic	0.74	0.78	0.74	0.69	0.81
	dup	1.03	2.33	1.03	1.01	4.42
	if	0.25	0.69	0.25	0.24	0.75
	ifcmp	4.05	3.83	4.05	3.96	4.20
	load	9.10	5.61	9.10	13.78	7.45
	load*n*	26.81	29.65	26.81	21.33	26.00
	store	0.51	0.55	0.51	4.48	0.92
	store*n*	1.39	2.68	1.39	1.34	4.72
JIT	Cycles	87766406	0.97	1.01	1.07	0.97
	SPARC instructions	66501513	0.97	1.00	1.06	0.95
	Instruction fetch stalls	1495605	1.10	0.99	0.92	0.80
	Data read misses	1282288	1.01	1.01	1.09	1.10
noJIT	Cycles	617569082	0.93	1.03	0.92	0.79
	SPARC instructions	462051681	0.91	1.03	0.92	0.77
	Instruction fetch stalls	19187351	1.01	1.00	1.03	1.04
	Data read misses	7198401	0.94	1.08	0.63	0.64
Toba	Cycles	188353979	1.01	0.98	0.96	0.99
	SPARC instructions	117678568	0.94	0.99	0.96	0.89
	Instruction fetch stalls	6328249	1.49	0.95	0.93	2.26
	Data read misses	1379597	0.89	0.92	0.92	0.87

5 CONCLUSIONS AND FUTURE WORK

Our results show that whole-program interprocedural optimizations can yield noticeable benefits on Java programs even in the face of hindrances like preexistence. Specifically, we were able to remove a significant number of methods calls and reduce the cost of those method calls that remain. What makes our optimization scheme unique is that it accounts for information introduced into the system at run-time by performing optimizations in the context of a persistent store.

We believe that a tighter coupling of the run-time system, persistent store, and optimizer will lead to greater performance enhancements. Optimization data such as method profile information, exact call site resolution, object creation information, and hardware behavior could be maintained as objects in a persistent store. As the persistent application executes and evolves the optimizer could make adjustments to the optimized code to reflect the state of the application.

BIBLIOGRAPHY

BIBLIOGRAPHY

AGESEN, O. 1994. Constraint-based type inference and parametric polymorphism. *Lecture Notes in Computer Science 864*, 78–100.

AGESEN, O. 1995. The cartesian product algorithm: Simple and precise typing of parametric polymorphism. Number 952 in Lecture Notes in Computer Science. Springer-Verlag, New York, NY, 2–26. ECOOP '95 – Object-Oriented Programming 9th European Conference, Aarhus, Denmark.

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. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 314–324.

ATKINSON, M. P., BAILEY, P. J., CHISHOLM, K. J., COCKSHOTT, P. W., AND MORRISON, R. 1983. An approach to persistent programming. *The Computer Journal 26*, 4 (Nov.), 360–365. Also published in/as: In "Readings in Object-Oriented Database Systems" edited by S. Zdonik and D. Maier, Morgan Kaufman, 1990.

ATKINSON, M. P. AND MORRISON, R. 1995. Orthogonally persistent object systems. *The VLDB Journal 4*, 3 (July), 319–401.

BACON, D. F. AND SWEENEY, P. F. 1996. Fast static analysis of C++ virtual function calls. In *Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '96)*. 324–341.

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 17th International Conference on Software Engineering*. 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 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), N. Meyrowitz, Ed. *SIGPLAN Notices 24*, 10 (Oct.), 49–70.

CHOW, F., CHAN, S., KENNEDY, R., LIU, S.-M., LO, R., AND TU, P. 1997. A new algorithm for partial redundancy elimination based on SSA form. In Proceedings of the ACM Conference on Programming Language Design and Implementation (Las Vegas, Nevada, June). *32,* 5 (May), 273–286.

DEAN, J., GROVE, D., AND CHAMBERS, C. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP'95—Object-Oriented Programming, 9th European Conference*, W. G. Olthoff, Ed. Lecture Notes in Computer Science, vol. 952. Springer, Aarhus, Denmark, 77–101.

DETLEFS, D. AND AGESEN, O. 1999. Inlining of virtual methods. In *Proceedings ECOOP'99*, R. Guerraoui, Ed. LCNS 1628. Springer-Verlag, Lisbon, Portugal, 258–278.

DIWAN, A., MOSS, J. E. B., AND MCKINLEY, K. S. 1996. Simple and effective analysis of statically-typed object-oriented programs. *ACM SIGPLAN Notices 31,* 10 (Oct.), 292–305.

ENBODY, R. 1998. *Perfmon User's Guide*. Michigan State University. http://web.cps.msu.edu/∼enbody/perfmon/index.html.

FERNÁNDEZ, M. F. 1995. Simple and effective link-time optimization of Modula-3 programs. *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. *ACM SIGPLAN Notices 30,* 10 (Oct.), 108–123.

HÖLZLE, U., CHAMBERS, C., AND UNGAR, D. 1992. Debugging optimized code with dynamic deoptimization. In *SIGPLAN '92 Conference on Programming Language Design and Implementation*. 32–43.

LINDHOLM, T. AND YELLIN, F. 1996. *The Java Virtual Machine Specification*. Addison-Wesley.

NYSTROM, N. 1998. Bytecode level analysis and optimization of java classes. M.S. thesis, Purdue University.

PALSBERG, J. AND SCHWARTZBACH, M. I. 1991. Object-oriented type inference. In *Proceedings OOPSLA '91, ACM SIGPLAN Notices*. 146–161. Published as Proceedings OOPSLA '91, ACM SIGPLAN Notices, volume 26, number 11.

PROEBSTING, T. A., TOWNSEND, G., BRIDGES, P., HARTMAN, J. H., NEWSHAM, T., AND WATTERSON, S. A. 1997. Toba: Java for applications – A way ahead of time (WAT) compiler. In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems* (Portland, Oregon, June). USENIX.

SPEC. 1998. Specjvm98 benchmarks. http://www.spec.org/osg/jvm98.

SUNDARESAN, V., RAZAFIMAHEFA, C., VALLE-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. In *Proceeings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, L. Meissner, Ed. ACM Sigplan Notices, vol. 34.10. ACM Press, N. Y., 292–305.

TOBA. 1998. Toba: A Java-to-C translator. http://www.cs.arizona.edu/sumatra/toba.

VANDRUNEN, T. J. 2000. A local optimization for stack-based architectures. Department of Computer Science, Purdue University.