Effective Interprocedural Optimization of Object-Oriented Languages

by

David Paul Grove

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

1998

Approved by ______________________________
(Chairperson of Supervisory Committee)

Program Authorized
to Offer Degree ______________________________

Date ______________________________

Doctoral Dissertation

University of Washington

Abstract

Effective Interprocedural Optimization of Object-Oriented Languages

by David Paul Grove

Chairperson of the Supervisory Committee: Professor Craig Chambers
Department of Computer Science and Engineering

This dissertation demonstrates that interprocedural analysis can be both practical and effective for sizeable object-oriented programs. Although frequent procedure calls and message sends are important structuring techniques in object-oriented languages, they can also severely degrade application run-time performance. A number of analyses and transformations have been developed that attack this performance problem by enabling the compile-time replacement of message sends with procedure calls and of procedure calls with inlined copies of their callees. Despite the success of these techniques, even after they are applied it is extremely likely that some message sends and non-inlined procedure calls will remain in the program. These remaining call sites can force an optimizing compiler to make pessimistic assumptions about program behavior, causing it to miss opportunities for potentially profitable optimizations. Interprocedural analysis is one well-known technique for enabling an optimizing compiler to more precisely model the effects of non-inlined calls, thus reducing their impact on application performance.

Interprocedural analysis of object-oriented and functional languages is quite challenging because message sends and/or applications of computed function values complicate the construction of the program call graph, a critical data structure for interprocedural analyses. Therefore, the core of this dissertation is an

in-depth examination of the call graph construction problem for object-oriented languages. It consists of the following components:

- A general parameterized algorithm that encompasses many well-known and novel call graph construction algorithms is defined.
- The general algorithm is implemented in the Vortex compiler infrastructure, a mature, multi-language, optimizing compiler. The Vortex implementation provides a "level playing field" for meaningful cross-algorithm performance comparisons.
- The costs and benefits of a number of call graph construction algorithms are empirically assessed by applying their Vortex implementation to a suite of sizeable (5,000 to 50,000 lines of code) Cecil and Java programs. Two small Smalltalk programs are also considered.

For many of the benchmark applications, interprocedural analysis enabled substantial speed-ups over an already highly optimized baseline. Furthermore, a significant fraction of these speed-ups can be obtained through the use of a scalable, near-linear time call graph construction algorithm.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I am deeply grateful to my advisor Craig Chambers for creating a fun and challenging environment in which to learn how to do research. During my first three years in the Cecil/Vortex group, I was lucky to have Jeff Dean as a friend, colleague, and late-night coding partner. I have many fond memories of discussing grand research ideas, Vortex implementation challenges, and the Sonics chances in the playoffs with Craig and Jeff over coffee in the Atrium or during one of our perhaps slightly overly competitive conference mini-golf games.

This dissertation was greatly improved by the careful reading and helpful comments of Susan Eggers, David Notkin, and Jeff Dean. They helped me clarify my thinking and remove a few "elephants from the closet."

Although our overly long lunch-time bull sessions and the infamous croquet and dart playing periods probably added months to all of our graduation dates, my years in graduate school were greatly enriched by the denizens of C110. Both the founding members: Anthony, Mike, Jeff, Neal, Wayne, Ted, and I, and the newer members: Sean, Jonathan, Jake, and Tashana have helped create a proud tradition of procrastination, coffee breaks, bagels, and tasteful decor. Over the years, Derrick, Sung, Ruth, AJ, and many of the other SPUDS, Disc Drive, and PromiseBreakers athletes have made my stay in Seattle much more enjoyable (after all, graduate school is really just an elaborate front for intramural sports). Special thanks to Jeff, Heidi, and Sean for their friendship through the years–it's made all the difference.

Finally, my deepest thanks to my family and my brothers in the Wimachtendienk who helped me learn all the truly important lessons long before I got to graduate school. None of this would have been possible without you.

Chapter 1

Introduction

Interprocedural analysis can be practical and effective for sizeable object-oriented programs.

Frequent procedure calls and message sends serve as important structuring techniques for object-oriented languages. However, these features can also severely degrade application run-time performance. This degradation is due both to the direct cost of implementing the call and to the indirect cost of missed opportunities for compile-time optimization of the code surrounding the call. A number of techniques have been developed to convert message sends into procedure calls (to *statically bind* the message send) and to inline statically bound procedure calls, thus removing both the direct and indirect costs of the call. However, even in moderately sized programs, after these techniques have been applied it is likely that some non-statically bound message sends and some non-inlined procedure calls will remain. If an optimizing compiler is forced to make pessimistic assumptions about the behavior of most non-inlined calls then it is likely that potentially profitable optimization opportunities will be missed leading to significant reductions in application performance.

Interprocedural analysis is one method for enabling an optimizing compiler to more precisely model the effects of non-inlined calls, thus enabling it to make less pessimistic assumptions about program behavior and reduce the performance impact of non-inlined call sites. An interprocedural analysis can be divided into two logically separate sub-tasks. First, the *program call graph*, a compile-time data structure that represents the run-time calling relationships among a program's procedures, is constructed. In most case this will be done as an explicit pre-phase before performing the "real" interprocedural analysis, although some analyses will interleave call graph construction and analysis and others may only construct the call graph implicitly. Second, the "real" analysis is performed by traversing the call graph to compute summaries of the effect of callees at each call site and/or summaries of the effect of callers at each procedure entry. These summaries are then consulted when compiling and optimizing individual procedures.

In strictly first-order procedural languages, constructing the program call graph is straightforward; at every call site the target of the call is directly evident from the source code. However in object-oriented languages or languages with function values, the target of a call cannot always be precisely determined solely by an examination of the source code of the call site. In these languages, the target procedures invoked at a call site are at least partially determined by the data values that reach the call site. In object-oriented languages, the method invoked by a dynamically dispatched message send depends on the class of the object receiving the message; in languages with function values, the procedure invoked by the application of a computed function value depends on the value of the function value. In general, determining the flow of values needed to build a useful call graph requires an interprocedural data and control-flow analysis of the program. But, interprocedural analysis in turn requires that a call graph be built prior to the analysis being performed. This circular dependency between interprocedural analysis and call graph construction is the key technical difference between interprocedural analysis of object-oriented and functional languages and interprocedural analysis of strictly first-order procedural languages. Effectively resolving this circularity is the key technical challenge of the call graph construction problem for object-oriented languages.

This dissertation demonstrates that interprocedural analysis can be both practical and effective for sizeable object-oriented programs. Because call graph construction is a key challenge in successfully applying interprocedural analysis to object-oriented programs, the core of the dissertation is an in-depth examination of the call graph construction problem for object-oriented languages. The main contributions of this dissertation are:

- A general parameterized algorithm for call graph construction is developed. The general algorithm provides a uniform vocabulary for describing call graph construction algorithms, illuminates their fundamental similarities and differences, and enables an exploration of the design space of call graph construction algorithms. The general algorithm is quite expressive, encompassing a spectrum of algorithms that ranges from imprecise near-linear time algorithms to a number of context-sensitive algorithms.

- The general algorithm is implemented in the Vortex compiler infrastructure. The description of the general algorithm naturally gives rise to a flexible implementation framework that enables new algorithms to be easily implemented and assessed. Implementing all of the algorithms in a single framework incorporated in a mature, multi-language, optimizing compiler provides a "level playing field" for meaningful cross-algorithm performance comparisons.

- A number of call graph construction algorithms are experimentally assessed. Call graph construction algorithms are assessed by comparing the costs of call graph construction, the relative precision of the call graphs produced by each algorithm, and the impact of call graph precision on the bottom-line effectiveness of interprocedural analysis. The Vortex implementation of the general algorithm is used to test a number of call graph construction algorithms on a suite of sizeable (5,000 to 50,000 lines of code) object-oriented programs written in Cecil and Java. Two small Smalltalk programs are also included. In comparison to previous experimental studies in this area, these experiments both cover a much wider range of languages and call graph construction algorithms and include applications that are an order of magnitude larger than the largest used in prior work. Assessing call graph construction algorithms on large programs is important because some of the algorithms, including one that previous work has claimed is scalable, cannot be practically applied to some of the larger programs in our benchmark suite. These experiments demonstrate that interprocedural analysis can be both computationally practical and effective for sizeable object-oriented programs: some scalable call graph construction algorithms produce call graphs of sufficient precision to enable interprocedural analysis to make substantial improvements in bottom-line application performance.

- For interprocedural analysis to be practical as part of a programmer's day-to-day development environment, it must be possible to correctly recompile the application after program changes, e.g. as part of the edit-compile-debug cycle, without reanalyzing and recompiling the entire program from scratch. Interprocedural analysis and subsequent optimization results in intermodule dependencies that preclude true separate compilation. Therefore, a design for integrating call graph construction and interprocedural analysis with Vortex's existing incremental recompilation infrastructure is proposed. One novel aspect of the design is its use of an implicit representation for several major categories of dependencies, thus substantially reducing expected dependency maintenance costs without impacting selectivity or invalidation processing costs.

The remainder of this dissertation is organized as follows:

- Chapter 2 reviews background material including a description of the general structure of the Vortex compiler infrastructure.
- Chapter 3 introduces the general parameterized algorithm for call graph construction and describes its implementation in the Vortex compiler.
- Chapter 4 utilizes the common vocabulary of the general call graph construction algorithm to uniformly describe a number of novel and well-known call graph construction algorithms. The algorithms are empirically assessed on a suite of Cecil, Java, and Smalltalk programs by using the Vortex implementation of the call graph construction framework.
- Chapter 5 addresses support for incremental recompilation after program changes; an essential part of making interprocedural analysis practical in the context of day-to-day program development.
- Finally, Chapter 6 concludes the dissertation and suggests some directions for future research.

Chapter 2

Background

This chapter reviews relevant background material. Section 2.1 discusses two programming language features that can complicate call graph construction: message sends and first-class functions. Section 2.2 surveys a number of techniques that can be applied to transform message sends into procedure calls, thus simplifying the call graph construction problem. Sections 2.3 and 2.4 provide an introduction to program call graphs and interprocedural analysis. Finally, section 2.5 highlights the relevant features of our experimental infrastructure, the Vortex compiler.

2.1 Challenging Language Features

To construct a program's call graph, a compiler must be able to determine which procedures[1] are invoked at each call site in the program. In a first-order procedural language, this task is straightforward: at each call site the name of the invoked (*callee*) procedure is evident from the source code. Any language feature that obfuscates this mapping from call sites to callee procedures complicates the call graph construction problem. All object-oriented languages contain at least one such obfuscatory language feature, dynamically dispatched message sends. Additionally, some object-oriented languages allow first-class function values. Both of these language features are arguably better in terms of programming methodology, but complicate the call graph construction problem by intertwining the program's data-flow and control-flow.

1. We will use the terms procedure, function, and method interchangeably. Both our call graph construction algorithms and the Vortex compiler treat them uniformly.

2.1.1 Message Sends

Message sends, also known as virtual function calls, generic function applications, or dynamic dispatches, are the primary mechanism for late binding in object-oriented programs. By inserting a level of indirection between the code that invokes an abstract operation on an object and the code that implements the invoked operation (the target method), message sends facilitate the creation of flexible, extensible, generic abstractions. When a message is sent during program execution, the target method of the message send is determined by dispatching on the class[2] of the object that receives[3] the message. Because the method invoked as a result of a message send depends on the class of the object that receives the message, a message send site can invoke any number of target methods thereby making it possible for a single piece of source code to correctly manipulate any concrete implementation of an abstract interface.

For example, to provide a set abstraction, a class library may define an abstract **`Set`** class and several concrete set implementations, e.g. **`HashSet`**, **`ListSet`**, and **`BitSet`**, that are all subclasses of **`Set`**. Programmers may either use the provided set implementations or create new set implementations by defining additional subclasses of **`Set`**. Each subclass of **`Set`** is required to define a few methods that implement primitive operations, e.g. **`addElement`** for element addition and **`do`** for iteration. Other set operations, such as element inclusion, set union, and set intersection, are defined once as methods of the **`Set`** class and inherited by the various subclasses of **`Set`**. A single **`union`** method can be applied to any pair of concrete set implementations because the **`union`** method relies solely on sending messages like **`do`** and **`addElement`** rather than on the details of any particular set implementation. Similarly, the set abstraction is generic

---

2. This dissertation often uses the term *class*. In prototype-based languages, such as Self [Ungar & Smith 87] and Cecil [Chambers 93], this term is not strictly correct; the analogous concept is that of all objects in a single clone family. However, for the purposes of call graph construction and other compile-time optimizations, these two concepts are interchangeable. Therefore we will uniformly use the more concise term: class.

3. In singly dispatched object-oriented languages, such as Smalltalk [Goldberg & Robson 83], C++ [Stroustrup 91], and Java [Gosling et al. 96], only the class of the first argument of the message send (commonly called the *receiver*) is used to determine the method invoked as a result of the message send. Multiply dispatched object-oriented languages, such as CLOS [Bobrow et al. 88], Cecil [Chambers 93], and Dylan [Shalit 96], generalize message dispatch by using the classes of any subset of the message send's arguments to determine the method invoked. All of the techniques discussed in this dissertation apply straightforwardly to multiply dispatched languages (and have been implemented for Cecil), however to simplify the exposition we will uniformly use the more concise terminology derived from singly dispatched object-oriented languages.

because it relies on sending the `=` message to compare pairs of elements rather than on any hard-wired definition of element equality.

For call graph construction, the challenge presented by message sends is that in general the callee procedure(s) at a call site cannot be precisely determined without first precisely determining the set of receiver classes at the call site. Although static type declarations may frequently help to narrow the set of possible receiver classes, and thus the set of possibly invoked methods, call graph construction algorithms that rely solely on the available static type information to resolve message sends can be too conservative to be effectively utilized for program optimization [Grove et al. 97].

2.1.2 First-class Functions

Most programming languages support some variant of first-class functions (the ability to create, manipulate, and invoke computed function values). This support may be somewhat limited, as in the case of procedure-valued parameters in Fortran and function pointers in C and C++, or it may have the full generality of anonymous lexically nested functions as found in object-oriented languages such as Smalltalk, Self, and Cecil, and in functional languages such as Scheme, Haskell, and ML. Just like message sends in object-oriented languages, in all of these languages the evaluation and application of a computed function value represents a form of late binding between the caller and the callee. The call graph construction problem is complicated in exactly the same fashion as with message sends: control-flow is dependent on the flow of function values through the program.

In fact, sending messages and invoking first-class functions are isomorphic. In languages with first-class functions, we can view each source-level occurrence of a function definition (or function whose address is taken) as creating a new class with a single method named **`apply`** whose body is the body of the function.[4] If the function was lexically nested within another function, then the class and its **`apply`** method are considered lexically nested in that same function. Evaluating the function definition or taking the address of a function are treated as instantiation sites of the new class, and invoking a function value is treated as sending the **`apply`** message to the object

4. A similar strategy is used by the Pizza implementation to translate closures into Java [Odersky & Wadler 97].

representing the function value. This encoding lets us focus solely on analyzing the flow of class instances through the program without any loss of generality.

2.2 Techniques for Optimizing Message Sends

A compiler can replace a dynamic dispatch with a static call whenever it can determine that a single method will be invoked for all possible receiver classes of that call site. A message send that is replaced by a call in this fashion has been *statically bound*. Static binding is primarily utilized to improve application run-time performance by reducing the frequency of expensive dynamic dispatches and enabling procedure inlining and subsequent optimization. However, it could also be utilized to transform a program prior to call graph construction to reduce the number of data-dependent call sites found in the program. A number of analyses and transformations have been developed to enable static binding; the subsequent subsections survey the main approaches.

2.2.1 Intraprocedural Class Analysis

Intraprocedural class analysis computes for each program expression a set of classes such that any run-time value of the expression is guaranteed to be an instance of one of the classes in the computed set [Johnson et al. 88, Chambers & Ungar 90]. The analysis can be defined as a standard iterative data-flow problem that maintains a mapping from variables to sets of classes, and propagates this mapping through the procedure's control-flow graph. By default, variables map to the set of all possible classes, but literals and the results of object allocations (**`new`**) are mapped to singleton class sets. Class sets are combined with set-union at control-flow merge points, and class sets are narrowed after a run-time class test conditional branch.

When a dynamically dispatched message send is encountered, this mapping is consulted to determine if the receiver of the message is known to be a single class, or a bounded union of classes. If only a single receiver class is possible, or if all classes in a union invoke the same method, then the message send can be statically bound. Because intraprocedural class analysis has only local knowledge, it cannot statically bind a message send if the receiver is known only to be an instance of class **`C`** or some subclass of **`C`**, since an unknown subclass of **`C`** may provide an overriding definition of the target method.[5] This limitation implies that knowing the static type of the receiver of a dynamically dispatched message, as is the case in statically typed languages like

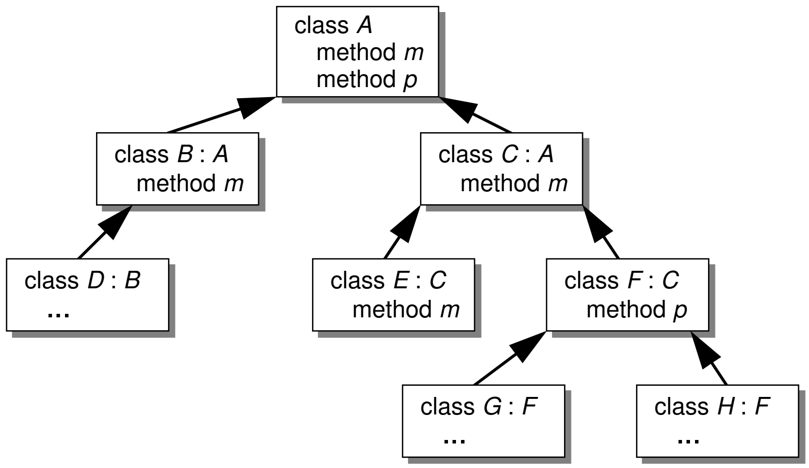


Figure 2.1: Class hierarchy analysis example

C++, Modula-3, and Java, is insufficient on its own to enable intraprocedural class analysis to statically bind the message send.

2.2.2 Class Hierarchy Analysis

Class hierarchy analysis [Fernandez 95, Dean et al. 95b, Diwan et al. 96] addresses one of the key weakness of intraprocedural class analysis by enabling the conversion of unbounded sets classes of the form "a variable holds an instance of some (unknown) subclass of **`C`**" into a bounded set of classes (only bounded sets of classes can provide useful information to the optimizer). Through this conversion, class hierarchy analysis enables the exploitation of static type declarations and specialized formal parameters. It does this by broadening the scope of the information available to the compiler by giving it access to all of the class and method declarations in the program. Given this global knowledge of the class hierarchy, the compiler can convert unbounded information of the form "a variable holds an instance of some (unknown) subclass of **`C`**" into a bounded set of classes.

For example, consider the class hierarchy shown in figure 2.1, where the method **`p`** in the class **`F`** contains a send of the **`m`** message to **`self`**. In general, sends of the **`m`** message must be

5. Unless the method is somehow annotated as non-overridable, e.g. Java's final.

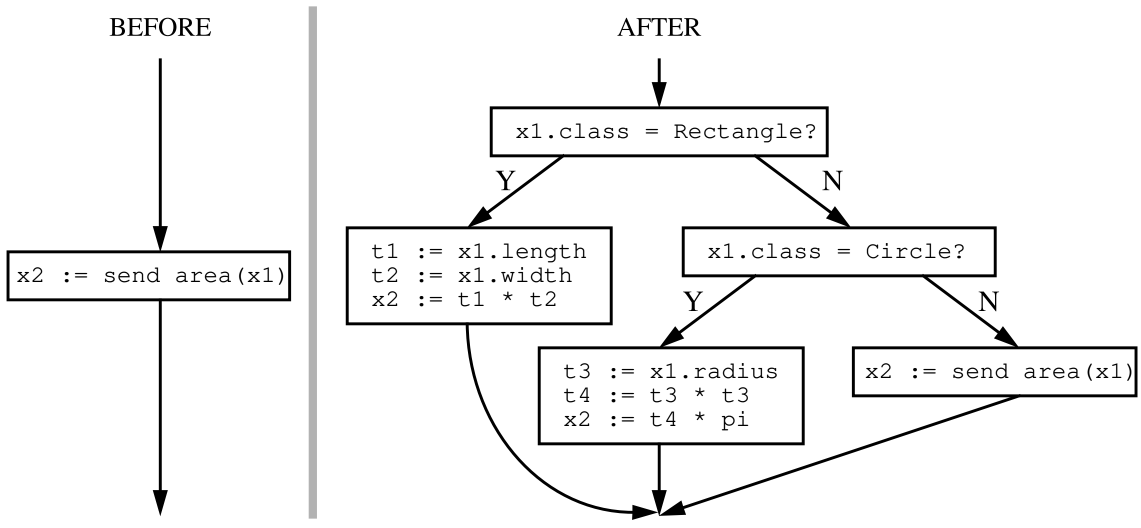


Figure 2.2: Control flow graph before and after insertion of class prediction tests

dynamically dispatched, since there are several implementations of **`m`** for subclasses of **`A`**. As a result, with only intraprocedural class analysis, the **`m`** message in **`F::p`** must be implemented as a general message send. However, by examining the subclasses of **`F`** and determining that there are no overriding implementations of **`m`**, the **`m`** message can be replaced with a direct procedure call to **`C::m`**. As this example illustrates, class hierarchy analysis can outperform source-level annotations such as C++'s **`non-virtual`** or Java's **`final`** because it can identify sub-trees of the hierarchy where a method that must be declared as virtual "reverts" to being non-virtual because there is only one possible implementation.

2.2.3 Receiver Class Prediction

There are message sends that cannot be statically bound solely through static analysis; some message sends *do* invoke more than one method at run-time. However, it is still possible to transform message sends of this type into a form that allows inlining of at least a subset of the possible target methods. As illustrated in figure 2.2, receiver class prediction is a simple local code transformation that converts a dynamic dispatch into a run-time class-case structure: one or more explicit in-line tests for particular expected classes, each of which branches when successful to a statically bound or inlined version of the target method for that class, possibly followed by a final dynamic dispatch to handle any remaining unpredicted classes. Receiver class prediction can be driven either by information hard-wired into the compiler, as in early Smalltalk and Self implementations [Deutsch & Schiffman 84, Chambers & Ungar 89], or by profile-derived class distributions [Calder & Grunwald 94, Hölzle & Ungar 94, Grove et al. 95], or by static examination of the program's class hierarchy [Chambers et al. 96b, Dean 96]. We use the term *profile-guided class prediction* to refer to class testing based on dynamic profile information and *static class testing* to refer to class-hierarchy-guided class testing.

If a large number of receiver classes are possible at a call site, testing for each individual class can be very expensive, especially in comparison to dispatch table based implementations of message sends. However, if the number of target methods is low, then run-time *subclass* tests can be inserted instead of class identity tests, leading to a number of run-time tests on the order of the number of possible methods rather than the number of possible classes.

It is quite common for a method to contain multiple message sends to a single receiver value; for example, several messages might be sent to **`self`** (or **`this`**). If receiver class prediction is applied to each of these message sends, redundant class tests will be introduced. Splitting can eliminate these redundant tests by duplicating paths through the control-flow graph starting at merges after one test and ending with the redundant test to be eliminated [Chambers & Ungar 90].

2.2.4 Customization and Specialization

Customization and method specialization [Chambers & Ungar 89, Lea 90, Dean et al. 95a] are techniques that also can be used to statically bind messages sent to a method's formal parameters. Specialization creates multiple copies of a single source method, each specialized to more precise tuples of receiver classes. Within each specialized method, the availability of more precise class information for some formal parameters enables the static binding of messages sent to those formal parameters.

2.3 Call Graphs

The call graph is a directed multi-graph that represents the calling relationship between a program's procedures. Call graphs will be formally defined in chapter 3; informally every

```
procedure main() {
   A();
   B();
}

procedure A() {
   Shape *s;
   s := new Rectangle(...);
   s->bounding_box()->draw();
}

procedure B() {
   Shape *s;
   s := new Circle(...);
   s->bounding_box()->draw();
}
```

```
Shape* Rectangle::bounding_box() {
      // return a Rectangle
}

void Rectangle::draw() {
   // draw a Rectangle on display
}

Shape* Circle::bounding_box() {
   // return a Rectangle
}

void Circle::draw() {
   // draw a Circle on display
}
```

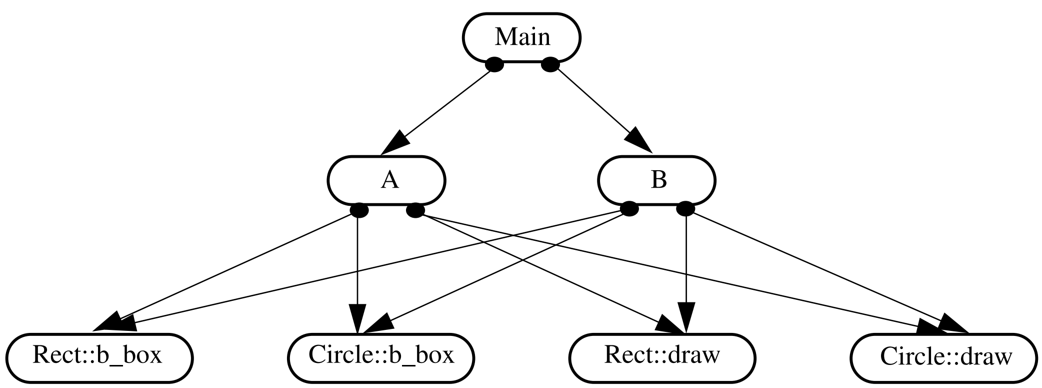


Figure 2.3: Example program and call graph

procedure in the program is represented by a node in the call graph. Each node has an associated collection of call sites, each of which contains edges to other nodes. An edge from a call site **`CS`** in node **`A`** to node **`B`** encodes that during some program execution procedure **`A`** may call procedure **`B`** from **`CS`**. If a call site is a direct procedure call, then there will be at most one call edge emanating from the call site. Indirect calls, either due to message sends or first-class functions, may result in call sites with multiple outgoing edges. Figure 2.3 contains an example program and a sound, but overly conservative, call graph for the program.

2.4 Interprocedural Analysis

Interprocedural analysis enables optimizing compilers to look across procedure call and message send boundaries, thus allowing them to make less pessimistic assumptions about program behavior and reduce the performance impact of non-inlined call sites [Spillman 71, Allen 74]. A number of interprocedural analyses have been defined; chapter 19 of Muchnick's book contains an overview of much of the prior work [Muchnick 97]. The program call graph must be constructed either prior to or in parallel with interprocedural analysis.

There are two main approaches to interprocedural analysis: supergraph-based analysis and summary-based analysis. In the first approach, analysis is performed over the program supergraph, also called the interprocedural control-flow graph (ICFG) [Weihl 80, Sharir & Pnueli 81, Landi & Ryder 91]. The ICFG is constructed by connecting the control-flow graphs of all procedures in the program with additional interprocedural control-flow edges from call sites to the corresponding callee procedure entries and from callee return sites back to the corresponding call sites (each such pair of interprocedural control-flow edges corresponds to one edge in the program call graph). After supergraph construction, interprocedural analysis simply consists of performing a standard intraprocedural data-flow analysis on the ICFG. In contrast, summary-based analyses never explicitly represent the entire program's control-flow. Instead, procedure-level summaries of the relevant data-flow facts are computed by a combination of local analysis and traversal of the program call graph. These summaries are typically stored in a program database, and are consulted as needed during the call graph traversal phase of interprocedural analysis and subsequent program compilation. Supergraph-based analyses may be more precise than summary-based analysis, but are typically more costly both in space and time and thus may not scale to large programs as gracefully as summary-based analyses. Program summary graphs [Callahan 88] represent an intermediate point between supergraph-based and summary-based analyses; they augment basic call graph based information with some information about intraprocedural flow-sensitive effects. The Vortex compiler implements a summary-based approach to interprocedural analysis.

Interprocedural analyses may be classified as *flow-sensitive* or *flow-insensitive* and as *context-sensitive* or *context-insensitive*. Generally speaking, supergraph-based analyses will be flow-sensitive, but summary-based analyses may be either flow-sensitive or flow-insensitive. Classifying an analysis as flow-sensitive or flow-insensitive is complicated by the lack of consensus in the literature on the exact definition of flow-sensitivity with respect to interprocedural analysis.[6] Part of the problem is that there are at least two distinct notions of flow-sensitivity, one based on control-flow and the other on data-flow. An analysis that computes program-point specific information is often called flow-sensitive to distinguish it from a (control) flow-insensitive analysis that assumes that statements within a procedure may execute in an arbitrary order. However, an analysis may also independently be data flow-sensitive or flow-insensitive; in a (data) flow-sensitive analysis information can only flow from the right-hand side to the left-hand side of an assignment, but in a (data) flow-insensitive algorithm information also flows from the left-hand side to the right-hand side of an assignment. To avoid ambiguity, this dissertation will label analyses as data flow-(in)sensitive or control flow-(in)sensitive. The distinction between context-sensitive and context-insensitive analysis is clearly defined, however. An analysis is context-insensitive if there is exactly one analysis-time representation of each procedure. If some notion of calling context is used to discriminate between multiple analysis-time representations of a procedure, thus enabling different analysis results to be computed for different calling contexts, then the analysis is context-sensitive.

2.5 Vortex Compiler Infrastructure

All of the empirical results in this dissertation are derived from experiments performed in the context of the Vortex compiler infrastructure. Figure 2.4 depicts the high-level structure of Vortex. Vortex is a language-independent, optimizing compiler back-end for object-oriented languages; currently, it supports front-ends for five object-oriented languages: Cecil, Smalltalk, Java, Modula-3 and C++. The experiments conducted for this dissertation make use of the Cecil, Smalltalk, and Java front-ends. All of the front-ends compile into the Vortex Intermediate Language (VIL), which is described in more detail elsewhere [Dean et al. 96]. All language-specific operations are localized to the various front-ends; all analyses and transformations in Vortex are performed in a language-independent fashion on VIL programs, thus ensuring that they are applied in the same, consistent manner across the different languages.

---

6. see [Marlowe et al. 95] for a detailed discussion of this issue.

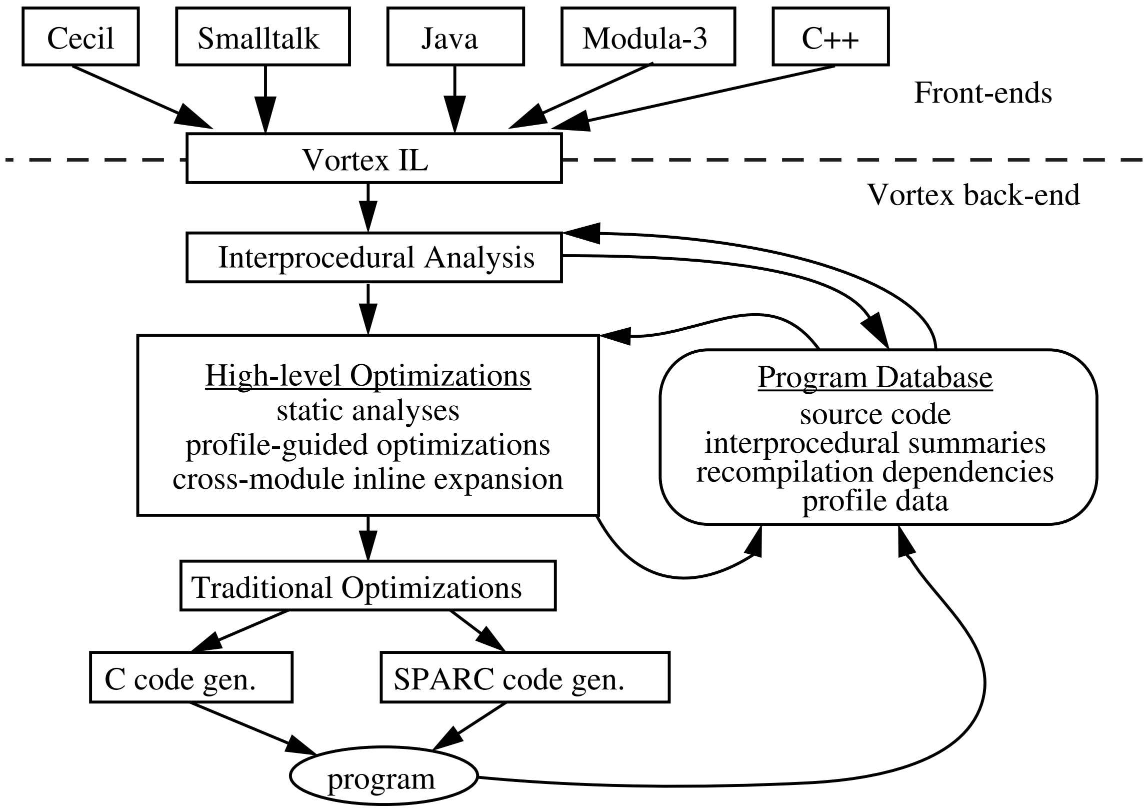


Figure 2.4: Vortex compiler architecture

After an input program is translated into VIL, interprocedural analysis is optionally performed and the resulting interprocedural summaries are saved in a persistent program database. The program is next compiled on a procedure-by-procedure basis. During compilation, each procedure passes through several stages. First, a suite of high-level optimizations are performed that use the techniques outlined in section 2.2 to statically bind message sends. Cross-module inline expansion of procedure calls is performed simultaneously with the static-binding transformations. Transformations are also performed to optimize away operations such as dead or partially dead closure and object creations. During this process, interprocedural summaries are consulted and non-local recompilation dependencies are recorded to enable incremental re-compilation after program changes. The high-level optimizations performed by Vortex are described in detail elsewhere [Chambers et al. 96b, Dean 96]. High-level VIL operations are then expanded into a series of lower-level operations, revealing additional opportunities for traditional optimizations such as common subexpression elimination (CSE). Vortex can then either generate portable C code or it can further lower the VIL program, perform low-level optimizations such as register allocation and instruction scheduling and generate SPARC assembly code.

Chapter 3

Call Graph Construction in Object-Oriented Languages

This chapter precisely defines both the call graph construction problem for object-oriented languages and a parameterized algorithm that solves the problem. Section 3.1 begins by illustrating the call graph construction problem and motivates a solution that integrates call graph construction with interprocedural class analysis. Section 3.2 precisely defines the output of instances of the parameterized call graph construction algorithm: program call graphs augmented with the class sets computed by interprocedural class analysis. The algorithm and its implementation in the Vortex compiler infrastructure are described in sections 3.3 and 3.4 respectively. The chapter concludes by discussing related work and summarizing its main results.

The parameterized call graph construction algorithm developed in this chapter is both a key technical result of this dissertation and a valuable expository device for succinctly defining particular call graph construction algorithms. By simply specifying the values with which to instantiate the general algorithm, a wide range of call graph construction algorithms can easily and precisely be defined. This method of defining call graph construction algorithms is used extensively in chapter 4, which presents an empirical evaluation of a number of algorithms.

3.1 The Role of Interprocedural Class Analysis

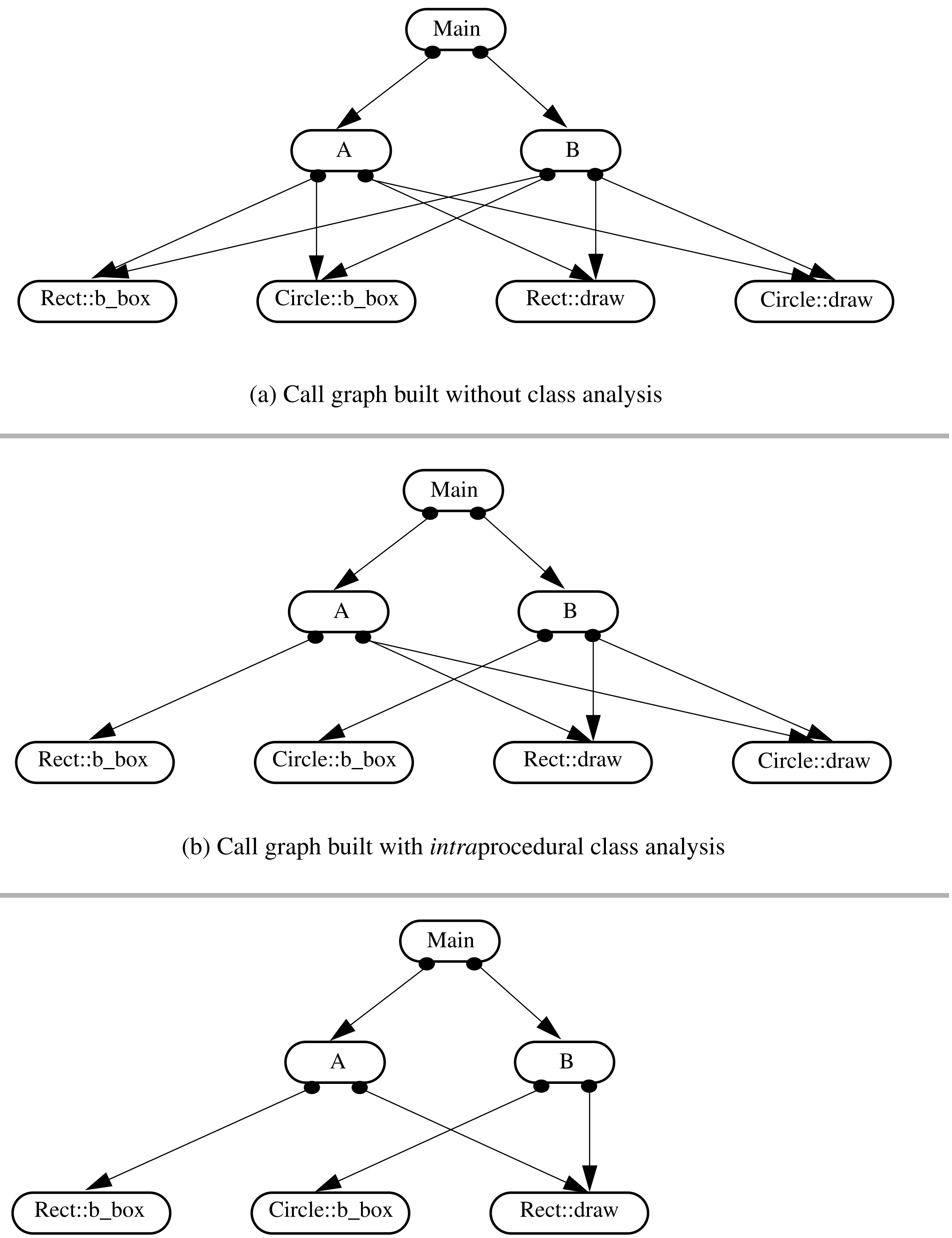
In object-oriented languages, the potential target method(s) of many calls cannot be precisely determined solely by an examination of the source code of their call sites. The problem is that the target methods that will be invoked as a result of the message send are determined by the classes of

```
procedure main() {
   A();
   B();
}

procedure A() {
   Shape *s;
   s := new Rectangle(...);
   s->bounding_box()->draw();
}

procedure B() {
   Shape *s;
   s := new Circle(...);
   s->bounding_box()->draw();
}

Shape* Rectangle::bounding_box() {
      // return a Rectangle
}

void Rectangle::draw() {
   // draw a Rectangle on display
}

Shape* Circle::bounding_box() {
   // return a Rectangle
}

void Circle::draw() {
   // draw a Circle on display
}
```

Figure 3.1: Example program

the objects that reach the message send site at run-time and thus act as receivers for the message. To illustrate this point, consider the small example program in figure 3.1. If only a combination of static type declarations and class hierarchy analysis is used during call graph construction to determine the possible callees at the various message send sites in the program, then the compiler must conservatively assume that at each call site any statically type-correct method could be invoked. This conservative assumption results in the call graph depicted in figure 3.2(a). However, an examination of the program reveals that this assumption results in an overly conservative call graph.[1] If the compiler also used class analysis to compute conservative approximations of the sets of possible receiver classes, then it could remove some of the spurious nodes and edges from the call graph. As a first step, the compiler could apply class analysis intraprocedurally to do a more precise job of tracking the local flow of classes from object creation points to message sends. For this program, this removes a spurious edge from the first call site in both the **`A`** and **`B`** procedures resulting in the call graph of figure 3.2(b). However, this call graph still contains some edges that cannot be present in any execution of the program. In particular, interprocedural class analysis can determine that both the **`Rectangle::bounding_box`** and **`Circle::bounding_box`**

1. Intuitively, an overly conservative call graph contains nodes and/or edges that do not correspond to some possible program execution, e.g. no possible execution of the sample program includes a call from procedure **`A`** to the **`Circle::draw`** method. Section 3.2 defines call graphs as a lattice domain and formalizes this notion.



(a) Call graph built without class analysis

(b) Call graph built with *intra*procedural class analysis

(c) Call graph built with *inter*procedural class analysis

Figure 3.2: Call graphs built with varying levels of class analysis

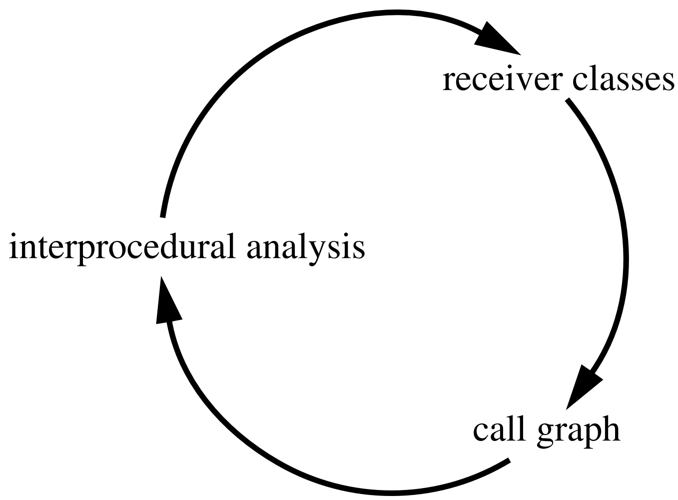


Figure 3.3: Circularity between call graph construction and interprocedural analysis

methods only return objects that are instances of the **`Rectangle`** class. Therefore, the edges to the **`Circle::draw`** method are spurious and can be removed, resulting in the call graph of figure 3.2(c).

As illustrated by this example, the presence of data-dependent calls in a program introduces a circular dependency between call graph construction and interprocedural analysis. Precisely determining the flow of values needed to build the call graph requires an interprocedural data and control-flow analysis of the program. But, interprocedural analysis in turn requires that a call graph be built prior to the analysis being performed. This circularity is illustrated in figure 3.3. Any call graph construction algorithm for object-oriented languages must address this circular dependency in one of two possible ways:

- by making a pessimistic (but sound) assumption: This approach breaks the circularity by making a conservative assumption for one of the three quantities and then computing the other two. For example, a compiler could perform no interprocedural analysis, assume that all statically type-correct receiver classes are possible at every call site, and in a single pass over the program construct a sound call graph (figure 3.2(a)). Similarly, intraprocedural class analysis could be performed within each procedure (making conservative assumptions about the interprocedural flow of classes) to slightly improve the receiver class sets before constructing the call graph (figure 3.2(b)). This overall process can be iteratively repeated to further improve the precision of the final call graph by using the current pessimistic solution to make a new, pessimistic assumption for one of the quantities and then using the new approximation to compute a better, but still pessimistic, solution. Although the simplicity of this approach is attractive, it may result in call graphs that are too imprecise to enable effective interprocedural analysis.

- by making an optimistic (but likely unsound) assumption and iterating to fix-point: Just like in the pessimistic approach, an initial guess is made for one of the three quantities giving rise to values for the other two quantities. The only fundamental difference is that because the initial guess may have been unsound, after the initial values are computed further rounds of iteration may be required to reach a fix-point. As an example, many call graph construction algorithms make the initial optimistic assumption that all receiver class sets are empty and that the main routine[2] is the only procedure in the call graph. Based on this assumption, the main routine is analyzed and in the process it may be discovered that in fact other procedures are reachable and/or some class sets contain additional elements, thus causing further analysis to be required. The optimistic approach can yield more precise call graphs (and receiver class sets) than the pessimistic approach, but is more complicated and may be more computationally expensive.

Chapter 4 presents and empirically assesses call graph construction algorithms that utilize both of the approaches. The data demonstrate that call graphs constructed by algorithms based on the first (pessimistic) approach are substantially less precise than those constructed by algorithms that utilize the second (optimistic) approach. Furthermore, these precision differences impact the effectiveness of client interprocedural analyses and in turn substantially impact bottom-line application performance. Therefore, in the remainder of this chapter we will concentrate on developing formalisms that naturally support describing optimistic iterative algorithms that integrate call graph construction and interprocedural class analysis (although the formalisms can also describe some non-iterative pessimistic algorithms).

---

2. For our purposes the *main routine* of a program consists of the union of all program entry points and static data initializations.

3.2 A Lattice-Theoretic Model of Call Graphs

This section precisely defines the output domain of the general integrated call graph construction and interprocedural class analysis algorithm. Section 3.2.1 informally introduces some of the key ideas. Section 3.2.2 formalizes the intuition of section 3.2.1 using ideas drawn from lattice theory. Finally, section 3.2.3 presents some applications of the call graph formalism by using it to discuss algorithmic termination, call graph soundness, and the relationship between the call graph formalism and the definition of the general algorithm presented in section 3.3.

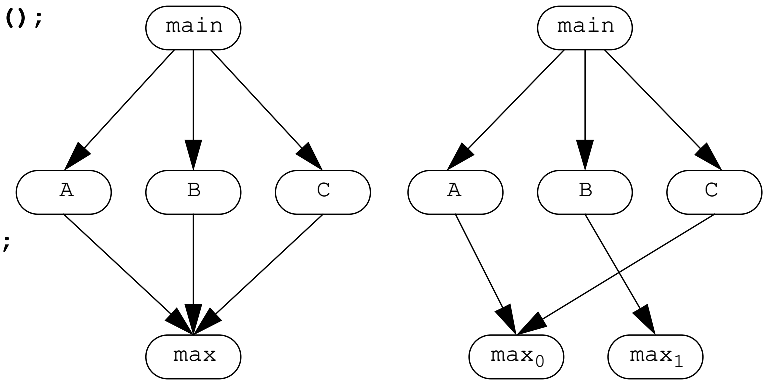
3.2.1 Informal Model of Call Graphs

As introduced in section 2.3, a call graph consists of nodes, representing procedures, that are linked by directed edges, representing calls from one procedure to another. However, this simple definition of call graphs is insufficient to accurately capture the output domains of context-sensitive interprocedural class analysis algorithms. Instead, each node will be defined to represent a *contour*: an analysis-time specialization of a procedure. In simple procedure-based call graphs, there is exactly one contour for each procedure; in contour-based call graphs, there may be an arbitrary number of contours representing different analysis-time views of a single procedure. Figure 3.4(b) shows the procedure-based call graph for the example program from figure 3.4(a). Figure 3.4(c) depicts the contour-based call graph constructed by a context-sensitive interprocedural class analysis algorithm that has separated the flow of integer and float objects by creating two contours for the `max` procedure.

Each contour intuitively contains five primary components:

- *Procedure identifier*: identifies which source-level procedure the contour is specializing.
- *Contour key*: encodes the context-sensitivity decisions made by the interprocedural class analysis algorithm; the actual contents of this component is algorithm-specific.
- *Class sets*: represent the result of interprocedural class analysis. Every contour contains class sets for formal parameters, local variables, and the result of the contour's procedure.

```
procedure main() {
   return A() + B() + C();
}

procedure A() {
   return max(4, 7);
}
procedure B() {
   return max(4.5, 2.5);
}
procedure C() {
   return max(3, 1);
}
```



(a) Example Program (b) Context-Insensitive (c) Context-Sensitive

Figure 3.4: Context-insensitive vs. context-sensitive call graph

These sets of classes represent the possible classes of objects stored in the corresponding variable (or returned from the procedure) during program execution.

- *Call graph edges*: record for each call site a set of possible callee contours.
- *Lexical parent contour(s)*: allow contours representing lexically nested procedures to access the class sets of free variables from the appropriate contour(s) of the enclosing procedure.

Intuitively, the first two components of a contour identify it. The third and fourth components record that portion of the final analysis results (class sets and call graph edges) that are local to the contour. The fifth component is only important for languages that allow lexically nested functions and is used to encode the lexical nesting relationship of contours. In addition to the “normal” contours created to represent procedures, special contours can be created to represent other program constructs. For example, a “root” contour can be created to represent the global scope: its local variables are the program’s global variables and its body is the program’s main routine. Modules can be similarly modeled by introducing one contour per module whose local variables are the module-scoped variables and whose body contains any module initialization code.

Interprocedural class analysis also needs to record sets of possible classes for each instance variable, in a manner similar to the class sets for local variables recorded in procedure contours.

Array classes are supported by introducing a single instance variable per array class to model the elements of the array. Instance variable contours intuitively consist of three main components:

- *Instance variable identifier:* identifies which source-level instance variable the contour is specializing.
- *Contour key*: just like the contour key component of procedure contours, this field encodes the context-sensitivity decisions made by the interprocedural class analysis algorithm.
- *Class set*: represents the potential classes of values stored in the instance variable.

Just like procedure contours, the components of an instance variable contour can be viewed as playing one of two roles: identifying the contour or recording local analysis results.

To analyze polymorphic data structures more precisely, some interprocedural class analysis algorithms introduce additional context-sensitivity in their analysis of classes and instance variable contents. Because array classes are modeled just like any other class, the analysis of polymorphic array classes can also benefit from any such context-sensitivity scheme. For example, by treating different instantiation sites of a class as leading to distinct (analysis-time) classes with distinct instance variable contours, an analysis can simulate the effect of templates or parameterized types without relying on explicit parameterization in the source program. Thus a single source-level class may be represented during analysis by multiple class contours. Class contours consist of two components:

- *Class identifier*: identifies the source-level class that the contour is specializing.
- *Contour key*: as with the other contour ley components, this field encodes context-sensitivity decisions made by the interprocedural class analysis algorithm.

All previously described class information, for example the class sets recorded in procedure and instance variable contours, is generalized to be class contour information.

Informally, the result of the combined call graph construction and interprocedural class analysis algorithm defined later in this chapter is a set of procedure contours and a set of instance variable contours. Together, the contents of these two sets define a contour call graph (the call graph edges component of all the procedure contours) and class contour sets for all interesting program constructs (the class set component of all the procedure and instance variable contours).

3.2.2 Formal Model of Call Graphs

This subsection uses lattice-theoretic ideas to formally define the contour-based model of context-sensitive call graphs described in the previous subsection. A lattice $D = \langle S_D, \leq_D \rangle$ is a set of elements S_D and an associated partial ordering $\leq_D$ of those elements such that for every pair of elements the set contains both a unique least-upper-bound element and a unique greatest-lower-bound element. A downward semilattice is like a lattice but only greatest-lower-bounds are required. The set of possible call graphs for a particular input program and call graph construction algorithm pair forms a downward semilattice; in this section the term "domain" will be used as shorthand for a downward semilattice. As is traditional in dataflow analysis [Kildall 73, Kam & Ullman 76] (but opposite to the conventions used in abstract interpretation [Cousot & Cousot 77]), if $A \leq B$ then B represents a better (more optimistic) call graph than A. Thus, the top lattice element, $\top$, represents the best possible (most optimistic) call graph, while the bottom element, $\perp$, represents the worst possible (most conservative) call graph. Not all elements of a call graph domain will be sound (safely approximate the "real" program call graph); section 3.2.3.2 formally defines soundness and some of the related structure of call graph domains.

3.2.2.1 Supporting Domain Constructors

The definition of the call graph domain uses several auxiliary domain constructors to abstract common patterns, with the intent of making it easier to observe the fundamental structure of the call graph domain. Some readers may want to skip ahead to section 3.2.2.2 to see how the constructors are used before reading the remainder of this section.

The constructor *Pow* maps an input partial order $D = \langle S_D, \leq_D \rangle$ to a lattice $DPS = \langle S_{DPS}, \leq_{DPS} \rangle$ where S_{DPS} is a subset of the powerset of S_D defined as: $S_{DPS} = \bigcup_{S \in PowerSet(S_D)} Bottoms(S)$ where $Bottoms(S) = \{ d \in S \mid \neg(\exists d' \in S, d' \leq_D d) \}$. The partial order $\leq_{DPS}$ is defined in terms of $\leq_D$ as: $dps_1 \leq_{DPS} dps_2 \equiv \forall d_2 \in dps_2,\ \exists d_1 \in dps_1$ such that $d_1 \leq_D d_2$. If S_1 and S_2 are both elements of S_{DPS}, then their greatest lower bound is $Bottoms(S_1 \cup S_2)$. *Pow* subtly differs from the standard powerset domain constructor, which maps an input set to a lattice whose domain is the full powerset of its input with a partial order based solely the subset relationship. The more complex definition of *Pow* serves to preserve the relationships established by its input partial order.

Intuitively, $Bottoms(S)$ serves to remove those elements which are redundant with respect to $\leq_d$ from S.

Each member of the family of constructors *kTuple,* $\forall k \geq 0$, is the standard *k*-tuple constructor which takes *k* input partial orders $D_i = \langle S_i, \leq_i \rangle$, $\forall i \in [1..k]$, and generates a new partial order $T = \langle S_T, \leq_T \rangle$ where S_T is the cross product of the S_i and $\leq_T$ is defined in terms of the $\leq_i$ pointwise $\langle d_{11}, \ldots, d_{k1} \rangle \leq_T \langle d_{12}, \ldots, d_{k2} \rangle \equiv \forall i \in [1..k],\ d_{i1} \leq_i d_{i2}$. If the input partial orders are downward semilattices, then *T* is also a downward semilattice; the greatest lower bound of two tuples is the tuple of the pointwise greatest lower bounds of their elements.

The constructor *Map* is a function constructor which takes as input a set *X* and a partial order $Y = \langle S_Y, \leq_Y \rangle$ and generates a new partial order $M = \langle S_M, \leq_M \rangle$ where $S_M = \{ f \subseteq X \times S_Y | (x, y_1) \in f \wedge (x, y_2) \in f \Rightarrow y_1 = y_2 \}$ and the partial order $\leq_M$ is defined in terms of $\leq_Y$ as $m_1 \leq_M m_2 \equiv \forall (x, y_2) \in m_2,\ \exists (x, y_1) \in m_1$ such that $y_1 \leq_Y y_2$. If the partial order *Y* is a downward semilattice, then *M* is also a downward semilattice; if m_1 and m_2 are both elements of S_M, then their greatest lower bound is $GLB_1 \cup GLB_2 \cup GLB_3$ where:

$$
\begin{aligned}
GLB_1 &= \{ (x, y) | (x, y_1) \in m_1, (x, y_2) \in m_2, y = glb(y_1, y_2) \} \\
GLB_2 &= \{ (x, y) | (x, y) \in m_1, x \notin dom(m_2) \} \\
GLB_3 &= \{ (x, y) | (x, y) \in m_2, x \notin dom(m_1) \}
\end{aligned}
$$

Finally, the constructor *AllTuples* takes an input downward semilattice $D = \langle S_D, \leq_D \rangle$ and generates a downward semilattice $V = \langle S_V, \leq_V \rangle$ by lifting the union of the *k*-tuple domains of *D*. The elements of S_V are $\perp$, all elements of $1Tuple(D)$, $2Tuple(D, D)$, $3Tuple(D, D, D)$, etc., and the partial order $\leq_V$ is the union of the individual *k*-tuple partial orders with the partial order $\{ (\perp, e) | e \in S_V \}$. If m_1 and m_2 are both elements of S_V and are both drawn from the same *k*-tuple domain of *D*, then their greatest lower bound is the greatest lower bound from that domain, otherwise their greatest lower bound is $\perp$.

3.2.2.2 Call Graph Domain

Figure 3.5 utilizes the domain constructors specified in the previous section to define the call graph domain for a particular input program and call graph construction algorithm pair. This definition is parameterized by inputs that encode program features and algorithm-specific context-sensitivity

CallGraph = *2Tuple(InstVarContourSet, ProcContourSet)*

InstVarContourSet = *Pow(InstVarContour)*

InstVarContour = *2Tuple(InstVarID, ClassContourSet)*

InstVarID = *2Tuple(InstVariable, InstVarKey)*

ProcContourSet = *Pow(ProcContour)*

ProcContour = *5Tuple(ProcID,*
Map(Variable, ClassContourSet), Map(CallSite, Pow(ProcID)),
Map(LoadSite, Pow(InstVarID)), Map(StoreSite, Pow(InstVarID)))

ProcID = *3Tuple(Procedure, ProcKey, Pow(AllTuples(ProcKey)))*

ClassContourSet = *Pow(ClassContour)*

ClassContour = *2Tuple(Class, Union(ClassKey, AllTuples(ProcKey)))*

Figure 3.5: Definition of call graph domain

polices. Program features are abstracted into seven unordered sets: *Class* is the set of source-level class declarations, *InstVariable* is the set of source-level instance variable declarations, *Procedure* is the set of source-level procedure declarations, *Variable* is the set of all program variable names, *CallSite* is the set of all program call sites, *LoadSite* is the set of source-level loads of instance variables, and *StoreSite* is the set of source-level stores to instance variables. The algorithm-specific information that encodes its context-sensitivity policies is represented by three partial orders: *ProcKey*, *InstVarKey*, and *ClassKey*. The *ProcKey* parameter defines the space of possible contexts for context-sensitive analysis of functions, i.e., procedure contours. The *InstVarKey* parameter defines the space of possible contexts for separately tracking the contents of instance variables, i.e., instance variable contours. The *ClassKey* parameter defines the space of possible contexts for context-sensitive analysis of classes, i.e., class contours. The ordering relation on these partial orders (and all derived domains) indicates the relative precision of the elements: one element is less than another if and only if it is less precise (more conservative) than the other.

The two components of a call graph are instance variable contours and procedure contours. Instance variable contours enable the analysis of dataflow through instance variable loads and stores, and procedure contours are used to represent the rest of the program. The components of these contours serve three functions:

- The first component of both instance variable and procedure contours serves to identify the contour by encoding the source level declaration the contour is specializing and the restricted context to which it applies. Each call graph is restricted to contain at most one *InstVarContour* for each *InstVarID* and at most one *ProcContour* for each *ProcID*. The third component of a *ProcID* identifies a chain of lexically enclosing procedure contours that is used to analyze references to free variables. For each procedure, the third component of all of its contour's *ProcID*'s is restricted to tuples of exactly length n, where n is the lexical nesting depth of the procedure.

- The second component of both instance variable and procedure contours records the results of interprocedural class analysis. In instance variable contours, it is simply a class contour set that represents the set of class contours stored in the instance variable contour. In procedure contours, it is a mapping from each of the procedure's local variables and formal parameters to a set of class contours representing the classes of values that may be stored in that variable. The variable mapping also contains an entry for the special token **`return`** which represents the set of class contours returned from the contour.

- The remaining components of a procedure contour encode the inter-contour flow of data and control caused by procedure calls, instance variable loads, and instance variable stores respectively. The third component, which maps call sites to elements of *Pow(ProcID)*, encodes the traditional notion of call graph edges.

The definition of *ClassContour* is somewhat complicated by the overloading of class contours to represent both objects and closure values. In both cases, the first component identifies the source-level class or closure associated with the contour. For classes, the second component of a class contour will contain an element of the *ClassKey* domain. For closures, the second component will contain a tuple of *ProcKey*'s that encode the lexical chain of procedure contours that should be used to analyze any references to free variables contained in the closure. This encoded information is used when procedure contours are created for the closure to initialize the third component of their *ProcID*'s.

For example, the 0-CFA algorithm is the classic context-insensitive call graph construction algorithm for object oriented and functional languages [Shivers 88, Shivers 91]. It can be modeled by using the single point lattice, $\{\bot\}$, to instantiate the *ProcKey*, *InstVarKey*, and *ClassKey* partial orders. Thus, each call graph in the call graph domain will have at most one procedure and instance variable contour. Another common context-sensitivity strategy is to create analysis-time specializations of a procedure for each of its call sites (Shivers's 1-CFA algorithm). This corresponds to instantiating the *ProcKey* partial order to the *Procedure* set, and using the single point lattice for *InstVarKey* and *ClassKey*. As a final example, the 1-CFA algorithm can be generalized to support the context-sensitive analysis of instance variables by tagging each class with the procedure in which it was instantiated and maintaining separate instance variable contours for each class contour. This context-sensitivity strategy is encoded by using the *Procedure* set to instantiate the *ProcKey* and *ClassKey* partial orders and elements of *ClassContour* as elements of the *InstVarKey* partial order.

3.2.3 Applications

The lattice-theoretic definition of call graphs is applied to describe conditions for algorithmic termination and call graph soundness. A final section discusses the relationships between the definition of the call graph domain and the call graph construction algorithm defined in section 3.3.

3.2.3.1 Termination

A call graph construction algorithm is *monotonic* if its computation can be divided into some sequence of steps, $S_1, S_2, \ldots, S_n$, where each S_i takes as input a call graph G_i and produces a call graph G_{i+1} such that $G_{i+1} \leq_{cg} G_i$ ($\leq_{cg}$ is the call graph partial order defined by the equations of figure 3.5). A call graph construction algorithm is *bounded-step* if each S_i is guaranteed to take a finite amount of time. If a call graph construction algorithm is both monotonic and bounded-step and its associated call graph lattice is of finite height,[3] then the algorithm is guaranteed to terminate in finite time. Furthermore, the worst-case running time of the algorithm is bounded by $O(LatticeHeight \times StepCost)$.

All of the sets specifying program features will be finite (since the input program must be of finite size), but the three algorithm-specific partial orders may be either finite or infinite. If the parameterizing partial orders are finite, then the call graph domain will have a finite number of

3. A lattice's height is the length of the longest chain of elements $e_1, e_2, \ldots, e_n$ such that $\forall i, e_i \leq e_{i-1}$.

elements and thus a finite height. This result follows immediately from the restriction on call graphs to contain at most one *InstVarContour* for each *InstVarID* and at most one *ProcContour* for each *ProcID*, the restriction that the third component of a ProcID will be a set of tuples of length exactly matching the lexical nesting depth of their procedure, and from the absence of any mutually recursive definitions in the equations of figure 3.5. However, some context-sensitive algorithms introduce mutually recursive definitions that cause their call graph domain to be infinitely tall. In these cases, care must be taken to incorporate a *widening* operation [Cousot & Cousot 77] to ensure termination. For example, the Cartesian Product [Agesen 95] and SCS [Grove et al. 97] algorithms described in section 4.4.1.1 both use elements of the *ClassContour* domain as part of their *ProcKey* domain elements. In the presence of closures (which are represented by the analysis as class contours), this can lead to an infinitely tall call graph lattice if a closure is recursively passed as an argument to its lexically enclosing procedure. Agesen terms this problem *recursive customization* and describes several methods for detecting it and applying a widening operation [Agesen 96].

3.2.3.2 Soundness

Figure 3.6 depicts the structure of the interesting portions of a call graph domain. If call graph *B* is more conservative than call graph *A*, i.e., $B \leq_{cg} A$ ($\leq_{cg}$ is the partial order over call graphs defined by the equations of figure 3.5), then *A* will be located above *B* in the diagram. The call graphs that exactly represent a particular execution of the program are located in the region labeled Optimistic. Because the call graph domain is a downward semi-lattice, we can define a unique call graph G_{ideal} as the greatest lower bound over all call graphs corresponding to a particular program execution. For a call graph to be sound, it must safely approximate any program execution, therefore G_{ideal} is the most optimistic sound call graph and a call graph *G* is sound iff it is equal to or more conservative than G_{ideal}, i.e., $G \leq_{cg} G_{\text{ideal}}$. Unfortunately, in general it is impossible to compute G_{ideal} directly, as there may be an infinite number of possible program executions, so this observation does not make a constructive test for the soundness of *G*. Note that not all call graphs are ordered with respect to G_{ideal}; figure 3.6 only depicts a subset of the elements of a call graph domain.

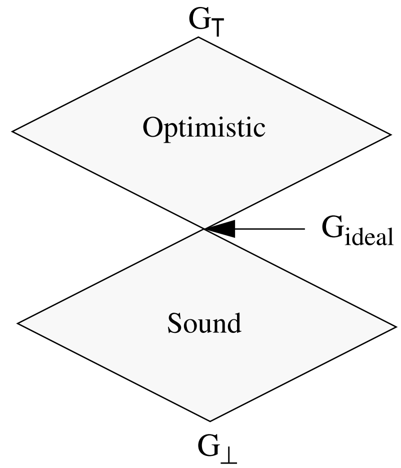


Figure 3.6: Regions in a call graph domain

3.2.3.3 Algorithm Definition

A final application of the call graph domain is the definition of the general integrated call graph construction and interprocedural class analysis algorithm found in the following section. The general algorithm is parameterized by context-sensitivity strategy functions whose codomains are *Pow(ProcKey)*, *Pow(InstVarKey)*, and *Pow(ClassKey)*. The result of applying the general algorithm to an input program is an element of the call graph domain implied by the input program and the codomains of the algorithm's context-sensitivity strategy functions. The computation of the algorithm can be viewed as a series of steps that transition an intermediate solution from one element of the algorithm/program call graph domain to another element. Finally, some of the core data structures used in the implementation of the algorithm are simply representations of the procedure, instance variable, and class contours defined in this section.

3.3 An Algorithmic Framework for Call Graph Construction

This section specifies the general integrated call graph construction and interprocedural class analysis algorithm for a small example language defined in section 3.3.1. The general algorithm is parameterized by four *contour key selection functions* that enable it to encompass a wide range of specific algorithms; the role of and requirements for these functions is explored in section 3.3.2.

```
Program      ::= {Decl} {Stmt} Expr
Decl         ::= TypeDecl | ClassDecl | InstVarDecl | VarDecl | MethodDecl
TypeDecl     ::= type TypeID {subtypes TypeID}
ClassDecl    ::= class ClassID  {inherits ClassID}
                                {subtypes TypeID} { {InstVarDecl} }
InstVarDecl ::= instvar InstVarID : TypeID
VarDecl      ::= var VarID : TypeID
MethodDecl   ::= method MsgID ({Formal}):TypeID { {VarDecl} {Stmt} Expr }
Formal       ::= FormalID @ ClassID : TypeID
Stmt         ::= VarAssign | IVarAssign
VarAssign    ::= VarID := Expr
IVarAssign   ::= Expr . InstVarID := Expr
Expr         ::= VarRef | IVarRef | NewExpr | ClosureExpr |
                  SendExpr | ApplyExpr
VarRef       ::= VarID | FormalID
IVarRef      ::= Expr . InstVarID
NewExpr      ::= new ClassID
ClosureExpr ::= lambda ({Formal}):TypeID { {VarDecl} {Stmt} Expr }
SendExpr     ::= send MsgID ( {Expr} )
ApplyExpr    ::= apply Expr ( {Expr} )
```

Figure 3.7: Abstract syntax for simple object-oriented language

Section 3.3.4 specifies the analysis performed by the algorithm using set constraints and section 3.3.5 discusses methods of constraint satisfaction.

3.3.1 A Simple Object-Oriented Language

The analysis is defined on the simple statically typed object-oriented language whose abstract syntax is given by figure 3.7.[4] It includes declarations of types, global and local mutable variables, classes with mutable instance variables, and multimethods; assignments to global, local, and instance variables; and global, local, formal, and instance variable references, class instantiation operations, closure instantiation and application operations, and dynamically dispatched message sends. The inheritance and subtyping hierarchies are separated to enable the modeling of languages such as Cecil that separate the two notions; languages with unified subtyping and inheritance hierarchy can be modeled by requiring that all type and inheritance declarations are parallel. Multimethods generalize the singly-dispatched methods found in many object-oriented

4. Terminals are in boldface, and braces enclose items that may be repeated zero or more times, separated by commas.

languages by allowing the classes of all of a message's arguments to influence which target method is invoked. A multimethod has a list of immutable formals. Each formal is specialized by a class, meaning that the method is only applicable to message sends whose actuals are instances of the corresponding specializing class or its subclasses. We assume the presence of a root class from which all other classes inherit, and specializing on this class allows a formal to apply to all arguments. Multimethods of the same name and number of arguments are related by a partial order, with one multimethod more specific than (i.e., overriding) another if its tuple of specializing classes is at least as specific as the other (pointwise). When a message is sent, the set of multimethods with the same name and number of arguments is collected, and, of the subset that are applicable to the actuals of the message, the unique most-specific multimethod is selected and invoked (or an error is reported if there is no such method). Singly dispatched languages can be simulated by not specializing (specializing on the root class) all formals other than the first, commonly called self, this, or the receiver in singly dispatched languages. Procedures can be modeled by methods none of whose formals are specialized. The language includes explicit closure instantiation and application operations. Closure application could be modeled as a special case of sending a message, as is actually done in Cecil and Smalltalk, but including an explicit application operation simplifies the specification of the analysis. Other realistic language features can be viewed as special versions of these basic features. For example, literals of a particular class can be modeled with corresponding class instantiation operations (at least as far as class analysis is concerned). Other languages features such as super-sends, exceptions and non-local returns from lexically nested functions can easily be accommodated, but are omitted to simplify the exposition. The actual implementation in the Vortex compiler supports all of the core language features of Cecil, and most of the core language features of Smalltalk and Java. Non-supported language features include reflective operations, such as dynamic class or method loading and perform-like primitives, and multithreading and synchronization.

We assume that the number of arguments to a method or message is bounded by a constant independent of program size, and that the static number of all other interesting program features (e.g., classes, methods, call sites, variables, statements, and expressions) is $O(N)$ where N is a measure of program size.

Procedure Key Selection Function (PKS):

$PKS(ProcContour, CallSite, AllTuples(ClassContourSet), Procedure) \rightarrow Pow(ProcKey)$

Instance Variable Key Selection Function (IVKS):

$IVKS(InstVariable, ClassContourSet) \rightarrow Pow(InstVarKey)$

Class Key Selection Function (CKS):

$CKS(Class, ProcContour) \rightarrow Pow(ClassKey)$

Environment Key Selection Function (EKS):

$EKS(Closure, ProcContour) \rightarrow Pow(AllTuples(ProcKey))$

Figure 3.8: Signatures of contour key selection functions

3.3.2 Algorithm Parameters

The general algorithm is parameterized by four *contour key selection functions* that collaborate to define the context-sensitivity polices used during interprocedural class analysis and call graph construction. The algorithm has two additional parameters, a constraint initialization function and a class contour set initialization function, that enable it to specialize its constraint generation and satisfaction behavior. By giving different values to these six strategy functions, the general algorithm can be instantiated to a wide range of specific call graph construction algorithms. The signature of the general algorithm is:

$$Analyze_{(PKS,\, IVKS,\, CKS,\, EKS,\, CIF,\, SIF)}(Program) \rightarrow CallGraph_{(PK,\, IVK,\, CK)}$$

The required signatures of the four contour key selection functions are shown in figure 3.8. These functions are defined over some of the constituent domains of the call graph domain, and their codomains are formed by applying the *Pow* domain constructor to the call graph domain's three parameterizing partial orders. Thus the contour key selection functions for an algorithm can be viewed as implying the call graph domains from which the result of an algorithm instantiation is drawn.

The particular roles played by each contour key selection functions are:

- The procedure contour key selection function (PKS) defines an algorithm's procedure context-sensitivity strategy. Its arguments are the contour specializing the calling procedure, a call site identifier, the sets of class contours being passed as actual parameters, and the callee procedure. It returns a set of procedure contour keys that indicate the contours of the callee procedure that should be used to analyze this call.
- The instance variable contour key selection function (IVKS) collaborates with the class contour key selection function to define an algorithm's data structure context-sensitivity strategy. IVKS is responsible for determining the set of instance variable contours that should be used to analyze a particular instance variable load or store. Its arguments are the instance variable being accessed and the class contour set of the load or store's base expression (the object through which the access is occurring). It returns a set of instance variable contour keys.
- The class contour key selection function (CKS) determines what class contours should be created to represent the objects created at a class instantiation sites. Its arguments are the class being instantiated and the procedure contour containing the instantiation site. It returns a set of class contour keys.
- The environment contour key selection function (EKS) determines what contours of the lexically enclosing procedure should be used to analyze any references to free variables contained in a closure. Its arguments are the closure being instantiated and the procedure contour in which the instantiation is being analyzed. It returns a set of tuples of procedure contour keys that encode the lexical nesting relationship. When a class contour representing a closure reaches an application site, this information is used to initialize the lexical parent information (the third component of the *ProcID*) of any contours created to analyze the application (see the *ACS* function of 3.9).

Contour key selection functions may ignore some (or all) of their input information in computing their results. The main restriction on their behavior is that contour selection functions must be monotonic[5] and that for all inputs their result sets must be non-empty.

5. A function F is monotonic iff $x \le y \Rightarrow F(x) \le F(y)$.

The general algorithm has two additional parameters whose roles are discussed in more detail in subsequent sections. The first of these, the constraint initialization function (CIF), allows the algorithm to choose between generating an equality, bounded inclusion, or inclusion constraint to express the relationship between two class contour sets (see section 3.3.3). The last parameter, the class contour set initialization function (SIF), allows the algorithm to specify the initial value of a class contour set (see section 3.3.5).

3.3.3 Notation and Auxiliary Functions

This section defines the notational conventions and auxiliary functions used in the algorithm specification of figure 3.10.

During analysis, sets of class contours are associated with every expression, variable (including formal parameters), and instance variable in the program. The class contour set associated with the program construct **PC** in contour κ is denoted by $[[\mathrm{PC}]]_{\kappa}$. The algorithm's computation consists of generating constraints that express the relationships among these class contour sets and determining an assignment of class contours to class contour sets that satisfies the constraints. The constraint generation portion of the analysis is expressed by judgements of the form $\kappa \vdash \mathrm{PC} \Rightarrow C$, which should be read as the analysis of program construct **PC** in the context of contour κ gives rise to the constraint set C. These judgements are combined in inference rules that informally can be understood as inductively defining the analysis of a program construct in terms of the analysis of its subcomponents. For example, the [Seq] rule of figure 3.10 describes the analysis of a sequence of statements in terms of the analysis of the individual statements:

$$\frac{\kappa \vdash \mathtt{S_1} \Rightarrow C_1 \qquad \kappa \vdash \mathtt{S_2} \Rightarrow C_2}{\kappa \vdash \mathtt{S_1;S_2} \Rightarrow C_1 \wedge C_2}$$

To analyze the program construct $\mathtt{S_1;S_2}$, the individual statements $\mathtt{S_1}$ and $\mathtt{S_2}$ are analyzed and any resulting constraints are combined.

The generalized constraints generated by the algorithm are of the form $A \supseteq_p B$ where p is a non-negative integer. The value of p is set by the algorithm's constraint initialization strategy function (CIF), and encodes an upper bound on how much work the constraint satisfaction sub-system is allowed to perform while satisfying the constraint. Section 3.4.4 discusses in more detail how the value of p influences constraint satisfaction in the Vortex implementation of the algorithm. The key idea is that the constraint solver has two basic mechanisms for satisfying the constraint that A is a superset of B; it can propagate class contours from B to A or it can unify A and B into a single set. The solver is allowed to attempt to propagate at most p classes from B to A before it is required to unify them. If $0 < p < \infty$, then the generalized constraint is a bounded inclusion constraint and will allow a bounded amount of propagation work to occur on its behalf. If $p = 0$, then the generalized constraint is an equality constraint and sets are unified as soon as a constraint is created between them. Finally, if $p = \infty$, then the generalized constraint is an inclusion constraint and will never cause the unification of the two sets. A generalized constraint may optionally include a filter set f, denoted $A \supseteq_p^f B$, which restricts the flow of class contours from B to A to those class contours whose *Class* component is an element of f. Filters are used to enforce restrictions on dataflow that arise from static type declarations and method specializers.

A number of auxiliary functions are used to concisely specify the analysis:

- Several of the helper functions are simply named k-tuple or map accessors that return sub-components of one of the k-tuple or map domains used to construct the call graph domain. *ID*(κ), and *Contents*(κ) access the *InstVarID*, and *ClassContourSet* components of the *InstVarContour* κ. Similarly, *ID*(κ) accesses the ProcID component of the *ProcContour* κ. *Proc*(*id*) accesses the *Procedure* component of the *ProcID id* and *Lex*(*id*) accesses the third (lexical chain) component of the *ProcID id*. Both *Var*(**V**, κ) and *Formal*(i, κ) are used to access pieces of the co-domain of the second (variable mapping) component of *ProcContour* κ; *Var* returns the class set associated with the formal or local variable **V** and *Formal* returns that of the i-th formal parameter.

- *Type*(**PC**) returns the static type of **PC**, *FormalDecl*(i, **M**) returns i-th formal of method **M**.

- Class hierarchy analysis is used to determine for each type **T** the set of classes that subtype **T**, denoted *Conformers*(**T**), and to determine for each class **C** the set of classes that inherit from **C**, denoted *SubClasses*(**C**).

- Finally, figure 3.9 defines four additional helper functions:

- *ExpVar* encapsulates the details of expanding references to free variables. It expands the lexical parent contour chain to find all procedure contours that should be used to analyze a reference to variable **v** made from procedure contour K

- *ICVS* determines the target contours for an instance variable load or store based on the class contour set of the base expression. It uses the algorithm-specific strategy function *IVKS*.

- *MCS* and *ACS* determine the callee contours for a message send or closure application based on the information currently available at the call site and the algorithm-specific strategy function *PKS*. Two helper functions, *Invoked* and *Applicable*, encapsulate the language's dispatching and application semantics. Based on the message name (or closure values) and the argument class contours, *Invoked* computes a set of callee procedures (or closures). Given a callee procedure and a tuple of argument class contours, *Applicable* returns a narrowed tuple of class contours that includes only those class contours that could legally be passed to the callee as arguments. The main difference between *MCS* and *ACS* is their computation of the encoded set of possible lexical parents for the callee contours. *MCS* simply uses the root contour, since the example language does not include nested methods. In contrast, *ACS* must extract the set of lexical chains from the second component of the closure class contours.

3.3.4 Algorithm Specification

Figure 3.10 defines the general algorithm by specifying for each statement and expression in the language the constraints generated during analysis; declarations are not included because they do not directly add any constraints to the solution (however, declarations are processed prior to analysis to construct the class hierarchy and method partial order). The algorithm assumes that the whole program is available for analysis; implications of this closed world assumption are discussed in chapter 5.

Static type information is used in the analysis of statements to ensure that variables are not unnecessarily diluted by assignments; sets of classes corresponding to right hand sides are filtered by the sets of classes that conform to the static type of left hand sides. This occurs both in the rules

$$ExpVar(\mathbf{V}, \kappa) = \{\kappa' \mid defines(\mathbf{V}, \kappa') \wedge (\kappa' \in enclosing(\kappa') \vee \kappa' = \kappa)\}$$

where $defines(\mathbf{V}, z)$ is true when $Proc(ID(z))$ is the procedure that defines $\mathbf{V}$

$$enclosing(x) = base \cup \bigcup_{x' \in base} enclosing(x')$$

$$base = \left\{ y \,\middle|\, \begin{array}{l} Proc(ID(y)) = LexEnclProc(Proc(ID(x))) \wedge \\ \langle Key(ID(y)), pk_1, \ldots, pk_n \rangle \in Lex(ID(x)) \wedge \\ Lex(ID(y)) = \langle pk_1, \ldots, pk_n \rangle \end{array} \right\}$$

$$IVCS(iv, base) = \{\kappa \mid InstVar(ID(\kappa)) = iv \wedge Key(ID(\kappa)) \in IVKS(iv, base)\}$$

$$MCS(\kappa, l, msg, args) = CC(\kappa, l, args, Invoked(msg, args), \{\langle root \rangle\})$$
$$ACS(\kappa, l, expr, args) = CC(\kappa, l, args, Invoked(expr, args), LC(expr))$$

where $LC(e) = \{lc \mid \exists cls \in e, Class(cls) \text{ is a closure} \wedge ClassKey(cls) = lc\}$

$$CC(\kappa, l, args, callees, lcs) = \bigcup_{p \in callees} \left\{ \kappa' \,\middle|\, \begin{array}{l} Proc(ID(\kappa')) = p \wedge \\ Key(ID(\kappa')) \in PKS(\kappa, l, Applicable(p, args), p) \wedge \\ Lex(ID(\kappa')) \in lcs \end{array} \right\}$$

Figure 3.9: Auxiliary functions

for explicit assignments, [VAssign] and [IVAssign], and in the implicit assignments of actuals to formals and return expressions to **`result`** in the [Send], [Apply], and [Body] rules. Even if all assignments in the source program are statically type-correct, this explicit filtering at assignment points can still be beneficial because some instantiations of the general algorithm may not be as precise as the language's static type system.

The [Prog] rule is the entry to the analysis; the top-level statements and expression are analyzed in the context of κ_{root}, the contour specializing the global scope. Statement sequencing, [Seq], is as expected: analysis of a sequence of statements entails adding the constraints generated by the analysis of each statement.

Assignment statements are handled by the [VarAssign] and [IVarAssign] rules. In both rules, the right hand side is analyzed, yielding some constraints, and a constraint is added from the set of class contours representing the right hand side, $[[\mathbf{E}]]_\kappa$, to each of the sets of class contours representing the left hand side. In the [VarAssign] rule, the left hand side class contour sets are computed by using the auxiliary function *ExpVar* to expand the encoded contour lexical parent chain. In the [IVarAssign] rule, the left hand side contours are computed by using the instance variable contour selector (IVCS). The [IVarAssign] rule also adds the additional constraints generated by analyzing the base expression of the instance variable access.

[Program]
$$\frac{\kappa_{root} \vdash \mathtt{S} \Rightarrow C_1 \qquad \kappa_{root} \vdash \mathtt{E} \Rightarrow C_2}{\vdash \mathtt{D\ S\ E} \Rightarrow C_1 \wedge C_2}$$

[Seq]
$$\frac{\kappa \vdash \mathtt{S_1} \Rightarrow C_1 \qquad \kappa \vdash \mathtt{S_2} \Rightarrow C_2}{\kappa \vdash \mathtt{S_1;S_2} \Rightarrow C_1 \wedge C_2}$$

[VAssign]
$$\frac{\kappa \vdash \mathtt{E} \Rightarrow C_1}{\kappa \vdash \mathtt{V:=E} \Rightarrow C_1 \wedge \bigwedge_{\kappa_i \in ExpVar(\mathtt{V},\kappa)} Var(\mathtt{V},\kappa_i) \supseteq_p^f [[\mathtt{E}]]_\kappa}$$
where $f = Conformers(Type(\mathtt{V}))$

[IVAssign]
$$\frac{\kappa \vdash \mathtt{B} \Rightarrow C_1 \qquad \kappa \vdash \mathtt{E} \Rightarrow C_2}{\kappa \vdash \mathtt{B.F:=E} \Rightarrow C_1 \wedge C_2 \wedge \bigwedge_{\kappa_i \in IVCS(\mathtt{F},[[\mathtt{B}]]_\kappa)} Contents(\kappa_i) \supseteq_p^f [[\mathtt{E}]]_\kappa}$$
where $f = Conformers(Type(\mathtt{F}))$

[VarRef]
$$\kappa \vdash \mathtt{V} \Rightarrow \bigwedge_{\kappa_i \in ExpVar(\mathtt{V},\kappa)} [[\mathtt{V}]]_\kappa \supseteq_p Var(\mathtt{V},\kappa_i)$$

[IVarRef]
$$\frac{\kappa \vdash \mathtt{B} \Rightarrow C_1}{\kappa \vdash \mathtt{B.F} \Rightarrow C_1 \wedge \bigwedge_{\kappa_i \in IVCS(\mathtt{F},[[\mathtt{B}]]_\kappa)} [[\mathtt{B.F}]]_\kappa \supseteq_p Contents(\kappa_i)}$$

[New]
$$\kappa \vdash \mathtt{new\ C} \Rightarrow [[\mathtt{new\ C}]]_\kappa \supseteq_p \{ClassContour(\mathtt{C}, key) \mid key \in CKS(\mathtt{C},\kappa)\}$$

[Closure]
$$\kappa \vdash \mathtt{cls} \Rightarrow [[\mathtt{cls}]]_\kappa \supseteq_p \{ClassContour(\mathtt{cls}, key) \mid key \in EKS(\mathtt{cls},\kappa)\}$$
where $\mathtt{cls} = \mathtt{lambda}_l\ \mathtt{(F):T\ \{\ D\ S\ E\ \}}$

[Send]
$$\frac{\forall i,\ \kappa \vdash \mathtt{E_i} \Rightarrow C_i \qquad \forall\ \kappa_j,\ \kappa_j \vdash Proc(\kappa_j) \Rightarrow C_j}{\kappa \vdash \mathtt{send\ Msg_l\ (E_1 \ldots E_n)} \Rightarrow \bigwedge_i C_i \wedge \bigwedge_{(\kappa_j,i)} \begin{pmatrix} Formal(i,\kappa_j) \supseteq_p^f [[\mathtt{E_i}]]_\kappa \wedge C_j \wedge \\ [[\mathtt{send\ Msg_l\ (E_1 \ldots E_n)}]]_\kappa \supseteq_p Var(\mathtt{result},\kappa_j) \end{pmatrix}}$$
where $\kappa_j \in MCS(\kappa, l, \text{Msg}, \langle [[\mathtt{E_1}]]_\kappa \ldots [[\mathtt{E_n}]]_\kappa \rangle),\ i \in \{1 \ldots n\}$
$f = Conformers(FormalDecl(i, Proc(\kappa_j))) \cap SubClasses(FormalDecl(i, Proc(\kappa_j)))$

[Apply]
$$\frac{\kappa \vdash \mathtt{E}_0 \Rightarrow C_0 \qquad \forall i,\ \kappa \vdash \mathtt{E_i} \Rightarrow C_i \qquad \forall\ \kappa_j,\ \kappa_j \vdash Proc(\kappa_j) \Rightarrow C_j}{\kappa \vdash \mathtt{apply}_l \mathtt{E}_0\ \mathtt{(E_1 \ldots E_n)} \Rightarrow C_0 \wedge \bigwedge_i C_i \wedge \bigwedge_{(\kappa_j,i)} \begin{pmatrix} Formal(i,\kappa_j) \supseteq_p^f [[\mathtt{E_i}]]_\kappa \wedge C_j \wedge \\ [[\mathtt{apply}_l \mathtt{E}_0\ \mathtt{(E_1 \ldots E_n)}]]_\kappa \supseteq_p Var(\mathtt{result},\kappa_j) \end{pmatrix}}$$
where $\kappa_j \in ACS(\kappa, l, [[\mathrm{E}_0]], \langle [[\mathtt{E_1}]]_\kappa \ldots [[\mathtt{E_n}]]_\kappa \rangle),\ i \in \{1 \ldots n\}$
$f = Conformers(FormalDecl(i, Proc(\kappa_j))) \cap SubClasses(FormalDecl(i, Proc(\kappa_j)))$

[Body]
$$\frac{\kappa \vdash \mathtt{S} \Rightarrow C_1 \qquad \kappa \vdash \mathtt{E} \Rightarrow C_2}{\kappa \vdash \mathtt{(F_1 \ldots F_n):T\ \{D\ S\ E\}} \Rightarrow C_1 \wedge C_2 \wedge Var(\mathtt{result},\kappa) \supseteq_p^f [[\mathtt{E}]]_\kappa}$$
where $f = Conformers(T)$

Figure 3.10: Specification of general algorithm

Basic expressions are handled by the next four rules. Variable and instance variable reference use their respective auxiliary functions to find a set of target contours, and then add constraints from the appropriate class contour set of each target contour to the set of class contours corresponding to the referencing expression ($[[\mathbf{V}]]_\kappa$ or $[[\mathbf{B.F}]]_\kappa$). As in the assignment rules, [IVarRef] also adds any constraints generated by analyzing the base expression (**B**) of the instance variable access. Analyzing a class instantiation, the [New] rule, entails adding a constraint from a set of class contours implied by the *ClassKeys* computed by the class contour key selection function to the set of class contours representing the new expression ($[[\mathbf{new\ C}]]$). The [Closure] rule is similar to the [New] rule, but it uses the environment contour key selection function to compute a set of tuples of *ProcKey* that encode the lexical chain of procedure contours that will be used to analyze references to free variables.

The constraints generated by the [Send] rule logically fall into two groups:

- The argument expressions to the send must be analyzed, and their constraints included in the constraints generated by the send ($\bigwedge_i C_i$).
- For each callee procedure contour, three kinds of constraints are generated:
    - actuals are assigned to formals: $Formal(i, \kappa_j) \supseteq_p^f [[\mathbf{E_i}]]_\kappa$,
    - the callee's body is analyzed: $\bigwedge_{\kappa_j} C_j$,
    - and a result is returned: $[[\mathbf{send\ Msg_1(E_1 \ldots E_n)}]]_\kappa \supseteq_p Var(\mathbf{result}, \kappa_j)$.

The auxiliary function *MCS* (message contour selector) is invoked during the analysis of a message send expression to compute the set of callee contours from the information currently available at the call site. As the available information changes, additional callee contours and constraints will be added. Thus, the constraint graph is lazily extended as new procedures and procedure contours become reachable from a call site. Call graph nodes and edges are created and added to the evolving solution as a side effect of calling *MCS*.

The analysis of closure applications is quite similar to that of message sends. The key differences are that the [Closure] rule must include the analysis of the function value ($\mathbf{E}_0$) and instead of the message contour selector, the apply contour selector (ACS) is invoked to compute the set of callee contours based on the information currently available at the call site.

Finally, the [Body] rule defines the analysis of the bodies of both methods and closures.

To allow varying levels of context-sensitivity to safely coexist in a single analysis, some additional constraints are required to express a global safety condition:

$$\forall \kappa_1, \kappa_2 \in \mathit{InstVarContour},\ \mathit{ID}(\kappa_1) \le \mathit{ID}(\kappa_2) \Rightarrow \mathit{Contents}(\kappa_1) \le \mathit{Contents}(\kappa_2)$$
$$\forall \kappa_1, \kappa_2 \in \mathit{ProcContour},\ \mathit{ID}(\kappa_1) \le \mathit{ID}(\kappa_2) \Rightarrow \mathit{ClassMap}(\kappa_1) \le \mathit{ClassMap}(\kappa_2)$$

The first rule states that if the identifiers of two instance variable contours are related, which implies that they are representing the same source level instance variable, then if the key of the first is at least as conservative as the key of the second, then the contents of the first must also be at least as conservative as the contents of the second. Similarly, if two procedure contours are representing the same procedure and the key of the first is at least as conservative as the key of the second, then the class set mapping of the first must also be at least as conservative as the class set mapping of the second. These constraints ensure that different degrees of context-sensitivity can coexist, while still ensuring that if a store occurs to a class set at one level of context-sensitivity, then it (or some more conservative class contour) appears in the corresponding class set of all more conservative views of the same source program construct. All of the algorithms described in chapter 4 trivially satisfy the second rule and only *k*-*l*-CFA for $l > 0$ does not trivially satisfy the first.

3.3.5 Constraint Satisfaction

Computing a final solution to the combined interprocedural class analysis and call graph construction problem is an iterative process of satisfying the constraints already generated by the analysis and adding new constraints to the constraint set as class contour sets grow and new procedures and/or procedure contours become reachable at call sites. A number of algorithms are known for solving systems of set constraints. Section 3.4 discusses the constraint satisfaction mechanisms used by the Vortex implementation framework. Aiken's survey paper reviews the definition of set constraints, outlines current research, open problems, and applications, and provides an excellent starting point for further reading in this area [Aiken 94].

In addition to the choice of solution algorithm, the initial values assigned to the sets can potentially have a large impact on both the time required to compute the solution and the quality (precision) of the solution. An algorithm's class contour set initialization function (SIF) determines the initial value assigned to all class contour sets other than those found on the right hand side of the constraints generated by the [New] and [Closure] rules (whose initial value is computed by the class key contour selection function). The most common strategy is to initialize all other class contour sets to be empty; this optimistic assumption will yield the most optimistic (most precise) final result. However, there are other interesting possibilities. For example, if profile-derived class distributions are available, then they could be used to seed class contour sets, possibly reducing the amount of time consumed by constraint satisfaction without negatively impacting precision. Another possibility is to selectively give pessimistic initial values in the hopes of greatly reducing constraint satisfaction time with only small losses in precision. For example, since it is common in large programs for polymorphic container classes, such as arrays, lists, and sets, to contain tens or even hundreds of classes, and since class set information tends to be most useful for program optimization when the cardinality of the set is small, an algorithm might initialize the class contour sets of all container classes' instance variable contours to bottom, i.e., the set of all classes declared in the program. This may result in faster analysis time, since the analysis of code manipulating container classes should quickly converge to its final (somewhat pessimistic) solution, without significant reductions in the bottom-line performance impact of interprocedural analysis. Also, some non-iterative pessimistic algorithms can be modeled by initializing all class contour sets to bottom.

3.4 Vortex Implementation

The goals of this section are to provide a high-level outline of the Vortex implementation, discuss its role as an implementation framework, highlight some of the design choices, and briefly describe the implementation of several key primitives. Aspects of the Vortex implementation of interprocedural class analysis and/or call graph construction have been described in several previous papers [Grove 95, Grove et al. 97, DeFouw et al. 98]. In particular, DeFouw et al. contains a more detailed description of the data structures used to implement equality and bounded inclusion constraints and analyzes their asymptotic costs.

3.4.1 Overview

The Vortex implementation of the general call graph construction algorithm closely follows the specification given in section 3.3. It is divided into two main subsystems: constraint satisfaction and constraint generation.

The core of the constraint satisfaction subsystem is a worklist-based algorithm that at each step removes a "unit of work" from the worklist and performs local propagation to ensure that all of the constraints directly related to that unit are currently satisfied. This unit of work may be either a single node in the dataflow graph or an entire procedure contour depending on whether the algorithm instance uses an explicit or implicit representation of the program's dataflow graph (section 3.4.3.1). Satisfying the constraints directly related to a single node simply entails propagating class contours as necessary to all of the node's immediate successors in the dataflow graph. Satisfying the constraints directly related to an entire procedure contour entails local analysis of the contour to reconstruct and satisfy the contour's local (intra-procedural) constraints and the propagation (as necessary) of the resulting class contour information along all of the contour's outgoing inter-contour (inter-procedural) dataflow edges, which may result in adding contours to the worklist.

The constraint generation subsystem is implemented directly from the specification in figure 3.10, with extensions to support Cecil, Java, and Smalltalk language features. A method is defined on each kind of Vortex AST node[6] to add the appropriate local constraints and to recursively evaluate any constituent AST nodes to generate their constraints. As implied by the *MCS* and *ACS* functions, constraints are generated lazily; no constraints are generated for a contour/procedure pair until the class contour sets associated with the procedure's formal parameters are non-empty, signifying that the contour/procedure pair has been determined to be reachable by the analysis (section 3.4.4.1 describes one efficient implementation of lazy growth of the constraint graph).

6. abstract syntax tree, a commonly used representation of a program. see [Aho et al. 86].

3.4.2 An Implementation Framework

The implementation of the general call graph construction algorithm consists of 9,500 lines of Cecil code. Approximately 8,000 lines of common code implement core data structures (call graphs, dataflow graphs, contours, class sets, etc.), the constraint satisfaction and generation subsystems, the interface exported to other Vortex subsystems, and abstract mix-in classes that implement common procedure, class, and instance variable contour selection functions. Instantiating this framework is straightforward; each of the algorithms described and empirically evaluated in chapter 4 is implemented in Vortex by 75 to 250 lines of glue code that combine the appropriate mix-in classes and resolve any ambiguities introduced by multiple inheritance. In addition to enabling easy experimentation, the implementation framework provides a "level playing field" for cross-algorithm comparisons. For all algorithms, the call graph and resulting interprocedural summaries are uniformly calculated and exploited by a single optimizing compiler. The algorithms all use the same library of core data structures and analysis routines, although depending on whether the algorithms use an implicit or explicit representation of the intraprocedural dataflow graph (discussed below) their usage of some portions of this library will be different. This flexibility is not free; parameterizability is achieved by inserting a level of indirection (in the form of message sends) at all decision points. However, I believe that this overhead only affects the absolute cost of call graph construction, not the relative cost of algorithms implemented in the framework or the asymptotic behavior of the algorithms.

3.4.3 Design Choices

3.4.3.1 Implicit vs. Explicit Dataflow Graphs

One of the most important considerations in the implementation of the framework was managing time/space trade-offs. Most previous systems explicitly construct the entire (reachable) interprocedural data and control flow graphs. Although this approach may be viable for smaller programs or simple algorithms, even with careful, memory-conscious, design of the underlying data structures, the memory requirements can quickly become unreasonable during context-sensitive analysis of larger programs. One feature of the Vortex implementation is the ability to allow algorithms to choose between using an explicit or an implicit representation of the program's dataflow graph. In the implicit representation, only those sets of class contours that are visible across procedure or instance variable contour boundaries (those corresponding to formal parameters, local variables that are accessed by lexically nested functions, procedure return values, and instance variables) are actually represented persistently. All derived class sets and all intra- and inter- procedural data and control flow edges are (re)computed on demand. This greatly reduces the space requirements of the analysis, but increases computation time since dataflow relationships must be continually recalculated and the granularity of reanalysis is larger. An additional limitation of the implicit dataflow graph is that it does not support the efficient unification-based implementation of equality and bounded inclusion constraints discussed in section 3.4.4.4. The Vortex implementations of those algorithms that only generate inclusion constraints (0-CFA, *k-l*-CFA, CPA, and SCS) utilize the implicit dataflow graph representation, because reducing their memory requirements was a primary goal.

3.4.3.2 Contour Management

The implementation associates a call graph node with each procedure (and closure) in the program. Each call graph node maintains a mapping from contour keys to contour objects that contain the class contour sets and call graph edges associated with that contour key/procedure pair. As analysis proceeds, new class contours and call graph edges are added to the information stored in contour objects until a sound fix-point is reached. In most analyses, once the procedure contour selection function creates a contour key (and thus a contour) to analyze a callee procedure at a call site, that contour will be a part of the final solution. However, the procedure contour selection functions used by some of the context-sensitive algorithms described in section 4.4.1.2 may actually remove contour keys from the set of callee contour keys used to analyze a call site and replace them with other more conservative contour keys (this is often done to reduce the cardinality of the callee contour set). When this happens, it may be the case that a contour key, and thus a contour object, is no longer being used in the analysis of any call site of its procedure. At this point, the now defunct contour could either be discarded as useless (thus reclaiming space) or preserved (possibly to be used again for the analysis of another call site). The current Vortex implementation always preserves such contours because reuse is fairly frequent. A better strategy might be to discard defunct contour objects using a moderately-sized software victim cache [Jouppi 90] to allow for some reuse while still reducing space costs.

3.4.3.3 *Iteration Order*

A key component of the constraint satisfaction sub-system is the worklist abstraction used to drive its iteration. Obvious implementations of a worklist include using recursion to obtain an implicit work stack, an explicit work stack, and an explicit work queue. For algorithms that use the implicit dataflow graph representation, and thus have a coarse-grained unit of work, the choice of a worklist implementation can have a large impact on analysis time. A stack-based implementation yields a last-in-first-out (LIFO) ordering of work, whereas a queue-based implementation yields a first-in-first-out (FIFO) ordering. Intuitively, a LIFO ordering has the advantage that the analysis of all callee contours is done before the analysis of the caller contour, thus ensuring that up-to-date sets of classes for the results returned from the callees are available to be used in the analysis of the caller. On the other hand, a FIFO ordering has the advantage of potentially batching multiple re-analyses of a contour. For example, a contour is initially enqueued for reanalysis because during the analysis of one of its callers it was determined that the argument class sets passed to the contour have grown, thus causing new elements to be added to the class contour sets representing the contour's formal parameters. While the contour is still enqueued, analysis of another one of its callers may result in an additional widening of the enqueued contour's formals. Both of these updates will be handled in a single reanalysis of the contour when it is finally reaches the front of the queue and is processed. Informal performance tuning revealed that both of these effects are important, and as a result the Vortex implementation actually uses a hybrid mostly-FIFO ordering. Its basic worklist is an explicit queue, but the first time a contour is encountered it is immediately analyzed via recursion rather than being enqueued for later analysis.

3.4.3.4 *Program Representation*

Interprocedural class analysis operates over a summarized AST representation of the program. The summarized AST abstracts the program by collapsing all non-object dataflow and by ignoring all intra-procedural control-flow. Non-object dataflow is the portion of a procedure's dataflow that is statically guaranteed to consist only of values that cannot directly influence the targets invoked from a call site, i.e., the values are not objects or functions and thus cannot be sent messages or applied. For example, an arbitrary side-effect free calculation using native (non-object) integers would be represented only by a reference to the integer AST summary node. The second part of summarization removes all side-effect free computations whose results are only used to influence control flow.

Because the summarized AST representation is control-flow-insensitive, summarizing all non-object dataflow cannot degrade analysis precision. However, using a control-flow-insensitive representation will at least theoretically result in less precise results than using a control-flow-sensitive representation. To assess the importance of intra-procedural control-flow-sensitivity for class analysis, we analyzed several Cecil and Java programs using both the summarized AST representation and a (control-flow-sensitive) control-flow graph representation. We found that there was no measurable difference in bottom-line application performance in programs analyzed with the two different representations, but that the AST-based analysis was roughly twice as fast because in Vortex the AST representation is cheaper to construct and analyze.

In addition to "normal" methods written in the source language, Vortex also allows programs to include primitive methods whose bodies may consist of either a textual form of Vortex's intermediate language (VIL) or an arbitrary fragment of C++ code that is copied as-is into the generated code. VIL primitives are analyzed to construct a summarized AST representation, and then are analyzed as if there were normal methods. Vortex cannot construct an AST representation of a C++ primitive, so it must instead rely on annotations written in a pragma language that captures a primitive's calling behavior, side-effects, and impact on class analysis.

3.4.3.5 Approximate Set Union Operations

Propagation of class contour information through the data flow graph is one of the main costs of interprocedural class analysis. This is especially true for algorithms that use an implicit representation of the dataflow graph since using a procedure contour as the unit of analysis may result in a large amount of unnecessary re-analysis each time the class contour set of a formal, free variable, or callee result changes. To reduce this problem, we exploit the observation that once several subclasses of a common parent class are elements of a given class contour set, then it is likely that other subclasses of the parent class will also eventually be added to the set. Therefore, if during a set union or element addition operation the cardinality of the result set exceeds a threshold value, then a compaction phase examines the set to see if any classes in the set share a common parent class. To preserve the most precise results while compacting, the candidate common parent that has the fewest number of subclasses not already included in the union is selected, and it and all of its subclasses are added to the set. This approximation reduces the size of the set representation (Vortex supports a compact "cone" representation for the class set corresponding to a class and all of its subclasses [Dean et al. 95b]) and may reduce the total number of times the set's contents change (by eagerly performing several subsequent class additions in a single step). In a previous version of the Vortex system, experiments on large Cecil programs showed that eager approximation reduced 0-CFA [Shivers 91] analysis time by a factor of 15 while only resulting in slowdowns of the resulting optimized executables of 2% to 8% [Grove et al. 97]. However, for small Cecil programs, in addition to causing larger performance degradations, this technique actually increased 0-CFA analysis time, because the overly conservative class sets led to the analysis of otherwise unreachable portions of the standard library.

3.4.4 Implementation of Key Primitives

3.4.4.1 Lazy Constraint Generation

The MCS and ACS functions ensure that all of the class contour sets representing the formal parameters of a contour are non-empty before the body of the procedure is analyzed in the context of the contour to generate constraints. A straightforward implementation of this requirement would be to explicitly check to see if all of the other formal class contour sets of a contour are non-empty each time an element addition causes a formal class contour set to transition from empty to non-empty; when the last such set became non-empty, the contour would be analyzed. An alternative, more efficient method is actually used in the Vortex implementation of the explicit dataflow graph. The dataflow edges corresponding to the actual-to-formal dataflow for a caller/callee pair are grouped together into a barrier. The barrier maintains a count of how many edges have class contours "blocked" on them waiting for the barrier to be broken; once all edges have blocked classes, the barrier is released and the callee contour it guards can be analyzed.

3.4.4.2 Filters

Several rules in the analysis restrict the flow of class contours through certain edges in the dataflow graph by interposing a filter class set derived from static type declarations and/or the dispatching semantics of the language. In the explicit dataflow graph representation, filters are implemented by optionally associating a filter class set (implemented as a bit set) with an edge in the dataflow graph; the filter prevents all class contours whose *Class* components are not an elements of the filter set from being propagated along the edge. In the implicit dataflow graph, filters are implemented using filter AST nodes. A filter AST node contains an expression sub-tree and a filter class set; the value of a filter AST node is the intersection of its filter class set with the class set produced as the value of its expression sub-tree.

3.4.4.3 Bounded Inclusion Constraints

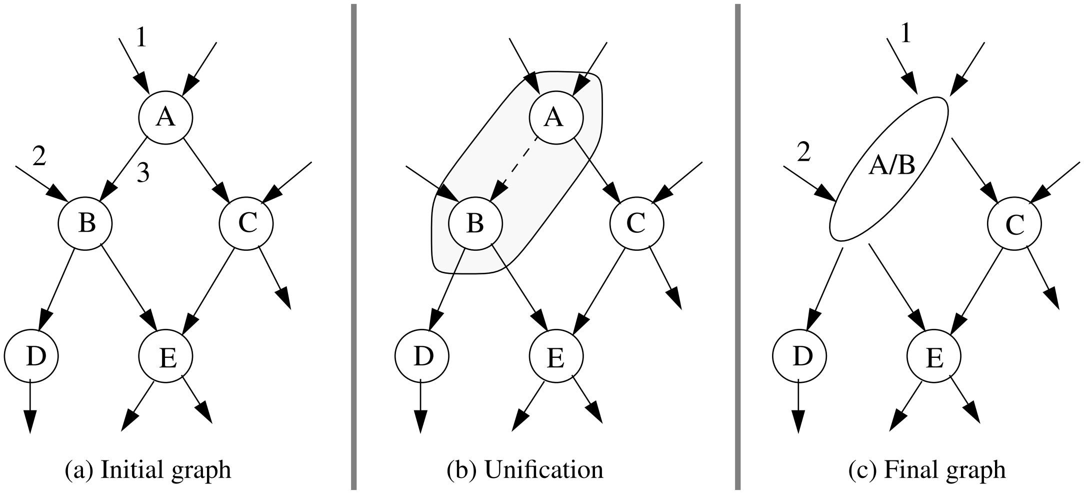
Equality constraints have been used to define near-linear time binding-time [Henglein 91] and alias [Steensgaard 96] analyses. In both of these algorithms, as soon as two nodes in the dataflow graph are determined to be connected, they are collapsed into a single node, signifying respectively that the source constructs represented by the two nodes either have the same binding-time or are potentially aliased. Although these algorithms are fast and scalable, they can also be quite imprecise. Bounded inclusion constraints attempt to combine the efficiency of equality constraints with the precision of inclusion constraints by allowing a bounded amount of propagation to occur across an inclusion constraint before replacing it with an equality constraint.

In the Vortex implementation, algorithms using either equality or bounded inclusion constraints must use the explicit dataflow graph representation. Each edge in the dataflow graph has a counter that is initialized to match the *p* value of its associated generalized inclusion constraint, $\supseteq_p$. Each time the constraint satisfaction sub-system attempts[7] to propagate a class contour across an edge, the edge's counter is decremented. When the counter reaches 0, the bounded inclusion constraint effectively becomes an equality constraint.

3.4.4.4 Satisfaction of Effective Equality Constraints

An *effective equality constraint* is either an equality constraint or a bounded inclusion constraint whose counter has been decremented to 0. Because the constraint satisfaction sub-system is not

7. The edge may have a filter which actually blocks the class contour from being propagated across the edge. Regardless of whether or not the class passes the filter, the edge's counter is decremented.



(a) Initial graph (b) Unification (c) Final graph

Figure 3.11: Unification

allowed to incur any propagation costs on the behalf of effective equality constraints, an alternative satisfaction method must be used. Therefore, once two nodes in the dataflow graph are linked by effective equality constraints, they are simply unified into a single node using fast union-find data structures [Tarjan 75]. Because unifying the nodes in the dataflow graph also causes the class contour sets associated with each node to be combined, any constraint between them will be satisfied without any additional work, because the two logical sets are now represented by the same actual set.

Figure 3.11 depicts the effect of unification on a portion of a program dataflow graph; nodes represent sets of class contours and directed edges between nodes indicate bounded inclusion constraints. In figure 3.11(b), propagation has occurred along the dashed edge between nodes **`A`** and **`B`** a sufficient number of times to trigger the unification of **`A`** and **`B`**; the resulting dataflow graph is shown in figure 3.11(c). In the final graph, class contours flowing along edge **`1`** into the new **`A/B`** node can be propagated to the original node **`B`**'s successors, **`D`** and **`E`**, without incurring the cost of crossing the now non-existent edge **`3`**. However, any class contours flowing along edge **`2`** into the new **`A/B`** node will be propagated to all three of its successor nodes, **`C`**, **`D`** and **`E`**. In the initial graph, class contours arriving at **`B`** via edge **`2`** could not reach node **`C`**; unification of **`A`** and **`B`** has potentially reduced the precision of the final results of the analysis.

3.4.4.5 Lazy Constraint Generation and Unification

Because constraints are generated lazily as procedures are proven to be reachable, the constraint generation and constraint satisfaction phases overlap. Therefore, it may become necessary to add a constraint $A \supseteq_p B$ when the "source" node B has already been collapsed out of the dataflow graph by unification. This can be done safely by adding a constraint $A \supseteq_p B_{unif}$, where B_{unif} is the node in the dataflow graph that is currently representing the set of unified nodes that includes B, and ensuring that all class contours currently in B_{unif}'s class contour set are propagated along the new edge. Our implementation actually takes a simpler approach and forcibly unifies A and all nodes downstream of A with B_{unif}. If there were no filter edges in the dataflow graph and all edges had uniform p values, then these two approaches would be equivalent. All of the algorithms that we have implemented so far do use uniform values for p, but since there will be filters on at least some edges in the graph, our forcible unification approach is overly pessimistic.

3.5 Related Work

This section discusses the relationship of the general call graph construction and interprocedural class analysis algorithm to other similarly general analysis frameworks. Individual algorithms for call graph construction, class analysis, or control flow analysis are presented and discussed throughout chapter 4.

Closely related to my analysis framework are four parameterized control flow analyses for higher-order functional languages [Stefanescu & Zhou 94, Jagannathan & Weeks 95, Nielson & Nielson 97, Palsberg & Pavlopoulou 98]. Like my framework, all of these frameworks are parameterized to allow them to express a variety of context-sensitive analyses. Because of their focus on higher-order functional core languages, the other four frameworks only have parameterized procedure contour selection function and do not directly address context-sensitive analysis of data structures. Thus, they are unable to express some of the algorithms, e.g. k-l-CFA with $l > 0$, that can be expressed in mine. Additionally, all four of these frameworks only utilize inclusion constraints and do not have a mechanism for choosing the initial value assigned to a class set; this prevents them from expressing unification-based algorithms (these algorithms utilize equality and bounded inclusion constraints) and pessimistic algorithms. On the other hand, the analysis framework of Nielson and Nielson includes rules for analyzing explicit let-bound polymorphism and thus can be instantiated to express polymorphic splitting [Wright & Jagannathan 98], an analysis which is not directly expressible in my analysis framework. The analysis framework of Nielson and Nielson also includes additional parameterization over environments that play a similar role as that of the class contour selection function at closure creation sites in our analysis. Perhaps the biggest difference between these other frameworks and mine is the context in which they are presented. Stefanescu and Zhou, Jagannathan and Weeks, and Nielson and Nielson all take a much more formal approach, defining a formal semantics for their language, specifying the analysis as an abstract interpretation of that semantics, and proving various properties of the analysis such as soundness and termination. The analysis framework of Palsberg and Pavlopoulou is intended as a formal tool for investigating the relationship between context-sensitive flow analysis and static type checking. In contrast, the primary purpose of the formal definition of our analysis framework is to enable a clear and precise definition of an algorithm; I have not formally proved any properties of the analysis. Unlike the other analysis frameworks, my framework is implemented in a mature, multi-language, optimizing compiler and is utilized as the basis for an extensive empirical evaluation of a number of call graph construction algorithms.

Agesen used *templates* as an informal explanatory device in his description of constraint-graph-based instantiations of several interprocedural class analysis algorithms [Agesen 94]. Templates are similar to contours in that they serve to group and summarize all of the local constraints introduced by a procedure. Agesen does not formally define templates and only considers context-sensitive analysis of procedures, not of instance variables or class instantiations.

3.6 Extensions and Future Work

The generalized algorithm of section 3.3 could be extended to a language that also included explicit let-bound polymorphism. Although none of the languages currently compiled by Vortex support let-bound polymorphism, it is possible that front-ends for functional languages such as ML [Milner et al. 90] or Scheme [Rees & Clinger 86] will eventually be added. By explicitly supporting let-bound polymorphism in the fashion of infinitary control flow analysis [Nielson & Nielson 97], the general analysis could encompass analyses such as polymorphic splitting [Wright & Jagannathan 98] that utilize let-expressions as syntactic clues to guided context-sensitivity decisions (see also section 4.8). The resulting combination of polymorphic splitting and bounded inclusion constraints would enable experimentation in a previously unexplored region of the call graph construction algorithm design space.

Two main improvements could also be made in the Vortex implementation framework. First, the unit of analysis in the implicit dataflow graph representation used by many of the context-sensitive analyses is the procedure contour. If any class contour set that was read during the analysis of the contour (and thus is linked by an inclusion constraint to a class contour set within the contour) changes, then the entire contour is re-analyzed. This may result in a large degree of wasted work if only a small part of the analysis of the contour actually depended on the modified class contour set. Program slicing [Weiser 84], reduction systems [Larchevêque 94] or jump functions [Callahan et al. 86, Grove & Torczon 93] could be used to enable the re-analysis of only those parts of the contour that actually depended on the modified class set without requiring an explicit representation of all intra-contour dataflow. Larchevêque actually used reduction systems to define an interprocedural class analysis algorithm, but because no experimental results are reported it is unclear how effective they would be at reducing analysis time.

Second, recent work on on-line cycle elimination in inclusion constraint graphs has demonstrated that orders of magnitude reductions in analysis time can be obtained in at least some circumstances [Fähndrich et al. 98]. Although their results do not explicitly apply to the call graph construction problem, the constraint graphs generated by many of the algorithms described in chapter 4 should be structurally similar to the inclusion constraint graphs generated by Andersen's alias analysis algorithm [Andersen 94] that were studied by Fähndrich et al. Therefore, it is likely that their on-line cycle elimination algorithm could usefully be applied to the inclusion constraint graphs generated by our call graph construction algorithm.

3.7 Summary

In object-oriented languages, constructing a precise call graph requires the integration of interprocedural class analysis and call graph construction. This chapter precisely defines the results of such integrated analyses, specifies an analysis framework that is parameterized to enable different context-sensitivity strategies, thus enabling a range of cost/precision trade-offs, and discusses the Vortex implementation of the framework. For each instantiation of the framework, analysis results (a call graph and a mapping from program constructs to sets of classes) are elements of a downward semi-lattice domain; the lattice-theoretic definition enables a precise definition of soundness and an understanding of the properties required to guarantee termination of the algorithm. By choosing different values for its contour selection functions and contour key partial orders, the general analysis specification can be instantiated to encode a wide range of call graph construction algorithms. The subsequent chapter describes a number of call graph construction algorithms as instantiations of the analysis framework developed in this chapter and empirically assess their performance on a suite of Cecil, Java, and Smalltalk programs.

Chapter 4

Assessing Call Graph Construction Algorithms

This chapter demonstrates that interprocedural analysis can be both effective and computationally practical for sizeable object-oriented programs. It examines six specific call graph construction algorithms that embody different precision/cost trade-offs, ranging from imprecise near-linear time algorithms to extremely precise but costly context-sensitive algorithms. Each algorithm is first specified as an instantiation of the general algorithm of chapter 3. Then the Vortex implementation of the algorithm is used to build a call graph that is used to perform several client interprocedural analyses; the resulting interprocedural summaries are exploited by Vortex's optimizer and the bottom-line performance of the resulting executable is empirically assessed. The experimental results demonstrate that some of these algorithms are both effective and practical: they produce call graphs that are sufficiently precise to enable subsequent interprocedural analyses to make substantial improvements in bottom-line application performance and are scalable in practice to sizeable object-oriented programs.

Section 4.1 details the experimental methodology used in this study by introducing the benchmark suite, describing how the experiments were performed, and defining the metrics used to assess the results. Next, section 4.2 presents experiments that establish upper and lower bounds on the expected impact of interprocedural analysis on application performance. The bulk of the chapter consists of sections 4.3 through 4.5 which each present a family of call graph construction algorithms using the following format:

- First, the algorithms are defined by viewing them as instances of the general call graph construction algorithm developed in the previous chapter. Instances of the general algorithm are distinguished by their choices of *ProcKey*, *ClassKey*, and *InstVarKey* partial orders, by their procedure, class, environment, and instance variable contour key selection functions, by their use of equality, bounded inclusion, or inclusion constraints, and by their choice of initial values for class contour sets.

- Second, the algorithms are experimentally assessed utilizing the Vortex compiler infrastructure. The results presented in this chapter focus on the following questions:
  - What is the relative precision of the call graphs produced by the algorithms?
  - What are the direct costs of using different call graph construction algorithms?
  - How do differences in call graph precision translate into differences in the effectiveness of client interprocedural analyses?

Section 4.3 covers 0-CFA [Shivers 88, Shivers 91], the classic context-insensitive call graph construction algorithm for object-oriented and functional languages. Section 4.4 focuses on context-sensitive algorithms that are more precise than 0-CFA, but may also be much more computationally expensive. It presents three context-sensitive algorithms: *k*-*l*-CFA [Shivers 88, Shivers 91], CPA [Agesen 95], and SCS [Grove et al. 97]. Finally, section 4.5 presents two families of algorithms, *p*-Bounded and *p*-Bounded Linear Edge [DeFouw et al. 98], that are less precise than 0-CFA, but are also substantially faster. The results of this chapter are summarized in two places: section 4.6, which recapitulates the definitions of the algorithms presented in this chapter and also discusses the relative precision of all the algorithms mentioned in this dissertation, and section 4.9, which concludes by comparing the bottom-line costs and benefits of a few representative algorithms. Section 4.7 presents some ideas for future work by highlighting some promising unexplored regions of the algorithmic design space and section 4.8 discusses other related work.

4.1 Experimental Methodology

This section describes the experimental methodology used to generate the data presented in this dissertation. Subsequent sections in this chapter present selected slices of the data; the complete data set can be found in appendix B.

Table 4.1: Brief description of benchmark programs

	Program	Lines[a]	Description
Cecil	richards	400	Operating systems simulation
	deltablue	650	Incremental constraint solver
	instr sched	2,400	Global instruction scheduler
	typechecker	20,000[b]	Typechecker for the old Cecil type system
	new-tc	23,500[b]	Typechecker for the new Cecil type system
	compiler	50,000	Old version of the Vortex optimizing compiler (circa 1996)
Java	cassowary	3,400	Constraint solver
	toba	3,900	Java bytecode to C translator
	java-cup	7,800	Parser generator
	espresso	13,800	Java source to bytecode translator[c]
	javac	25,500	Java source to bytecode translator[c]
	pizza	27,500	Pizza compiler
Smalltalk	richards	695	Operating systems simulation
	deltablue	1,154	Incremental constraint solver

a. Excluding standard libraries. The Cecil versions of richards and deltablue include a 4,850-line subset of the standard library; all other Cecil programs include the full 10,700-line standard library. All Java programs include a 16,400-line standard library. For the Smalltalk programs, line counts represent filed-out sources. The Smalltalk programs include portions of the ParcPlace Smalltalk standard library, although determining exactly how many lines of code this represents is not straightforward both because of the lack of "real" source files and because the Vortex Smalltalk front-end includes a treeshaking optimization that enables it to reduce the number of classes and methods actually contained in the VIL files it generates. Despite this optimization, large portions of the standard library are included due to library initialization code. The VIL files for the Smalltalk richards benchmark contain 64,000 lines; for comparison the VIL files for java-cup contain 68,000 lines.

b. The two Cecil typecheckers share approximately 15,000 lines of common support code, but the type checking algorithms themselves are completely separate and were written by different people.

c. The two Java translators have no common non-library code and were developed by different people.

4.1.1 Benchmark Description

Experiments were performed on the suite of object-oriented programs described in table 4.1; a more detailed description of the benchmark applications can be found in appendix A. With the exception of the Smalltalk programs and two of the Cecil programs, all of the applications are substantial in size. Previous experimental assessments of call graph construction algorithms for object-oriented or functional languages have almost exclusively used benchmark programs

Table 4.2: Language characteristics

Language	Object Model	Typing	All Methods Dispatched?	Function Values?	Multi-Methods?
Cecil	Pure	Dynamic[a]	Yes	Yes	Yes
Java	Hybrid	Mostly static	No[b]	No	No
Smalltalk	Pure	Dynamic	Yes	Yes	No

a. Cecil allows mixing statically and dynamically typed code, and running the static typechecker is optional. As a result, the optimizer ignores static type declarations and ensures type-safety through dynamic checks where needed.

b. **`final`** methods cannot be overridden, although they can override other methods. Our Java front-end resolves and statically binds all message sends where static type information indicates that only a single **`final`** method could be invoked. Invocations of static class methods are also statically bound by the front end.

consisting of only tens or hundreds of lines of source code; only a few of the previous studies included some larger programs that ranged in size up to a few thousand lines of code. The experimental results in this dissertation improve on previous results by considering programs that an order of magnitude larger than those used in prior work. Program size is an important issue, because many call graph construction algorithms have worst-case running times that are polynomial in the size of the program. The data demonstrates that these bounds are not just of theoretical concern; in practice many of the algorithms exhibit running times that appear to be super-linear in program size, and thus in fact cannot be successfully applied to programs of more than a few thousand lines of code. Conversely, all of the algorithms are quite successful at analyzing programs of several hundred lines of code, suggesting that benchmarks need to be a certain minimal size (and contain enough interesting polymorphic code) before they are useful tools for assessing call graph construction algorithms.

The benchmark suite contains programs written in three different object-oriented languages, allowing us to investigate the impact of programming language features on the costs and benefits of interprocedural analysis. Table 4.2 summarizes several distinguishing language features. In a pure object-oriented language, even the most primitive operations are invoked by sending messages. Therefore we expect that the call graph construction problem would be harder in Cecil and Smalltalk than in Java since message sends will be more frequent and source-level polymorphism is much more frequent. Additionally, both Cecil and Smalltalk programs make heavy use of lexically nested first-class functions, further complicating the call graph construction problem. We also expect that because of the higher frequency of message sends, the potential benefits of static binding, inlining, and other interprocedural analyses will be larger for Cecil and Smalltalk than for Java. Static type declarations are excellent fodder for class hierarchy analysis, so one would expect it to be more effective in Java, thus further lowering the potential additional performance benefits from more aggressive interprocedural analyses.

4.1.2 Experimental Setup

All experiments were conducted using the Vortex compiler infrastructure. The basic methodology will be to augment an already optimized base configuration with several interprocedural analyses performed over the call graph constructed by one of the algorithms. Because profile-guided class prediction (pgcp) [Hölzle & Ungar 94, Grove et al. 95] has been demonstrated to be an important optimization for some object-oriented languages, but may not be desirable to include in every optimizing compilers, all of the experiments use two base configurations: one purely static and one with profile-guided class prediction. Benchmark programs were compiled using the following configurations:

- The **base** configuration represents an aggressive combination of intraprocedural and limited interprocedural optimizations which include: intraprocedural class analysis [Johnson et al. 88, Chambers & Ungar 90], hard-wired class prediction for common messages (Cecil and Smalltalk only) [Deutsch & Schiffman 84, Chambers & Ungar 89], splitting [Chambers & Ungar 89], class hierarchy analysis [Fernandez 95, Dean et al. 95b, Diwan et al. 96], inlining, static class prediction [Dean 96], closure optimizations that identify and stack allocate LIFO-closures and sink partially-dead closure creations (Cecil and Smalltalk only), and a suite of traditional intraprocedural optimizations such as common subexpression elimination, constant propagation and folding, dead assignment elimination, and redundant load and dead store elimination.
- The **base+pgcp** configuration augments **base** with profile-guided class prediction [Hölzle & Ungar 94, Grove et al. 95]. For all programs with non-trivial inputs, different input sets were used to collect profile data and to gather other dynamic statistics (such as application execution time).

- For each call graph construction algorithm *G*, the **base+IP$_G$** configuration augments **base** with the following interprocedural analyses that enable the intraprocedural optimizations included in **base** to work better:

  - *Class analysis*: As a side-effect of constructing the call graph, each formal, local, global, and instance variable is associated with a set of classes whose instances may be stored in that variable. Intraprocedural class analysis exploits these sets as upper bounds that are more precise than “all possible classes,” enabling better optimization of dynamically dispatched messages.

  - *MOD analysis*: This interprocedural analysis computes for each procedure a set of global variables and instance variables that are potentially modified by calling the procedure. A number of intraprocedural analyses exploit this information to more accurately estimate the potential effect of non-inlined calls on local dataflow information.

  - *Exception detection*: This interprocedural analysis identifies those procedures which are guaranteed to not raise exceptions during their execution. This information can be exploited both to streamline their calling conventions and to simplify the intraprocedural control flow downstream of calls to exception-free routines.

  - *Escape analysis*: Interprocedural escape analysis identifies first-class functions which are guaranteed not to out-live their lexically enclosing environment, thus enabling the function objects and their environments to be stack-allocated [Kranz 88]. The analysis could be generalized to enable the stack allocation of objects as well, but the current Vortex implementation only optimizes closures and environments, and thus escape analysis only applies to the Cecil and Smalltalk benchmarks.

  - *Treeshaking*: As a side-effect of constructing the call graph, the compiler identifies those procedures which are unreachable during any program execution. The compiler does not compile any unreachable procedures, often resulting in substantial reductions both in code size and compile time.

- The **base+IP$_G$+pgcp** configuration augments the **base+IP$_G$** configuration with profile-guided class prediction. We used the same dynamic profile data (derived by iteratively optimizing, profiling, and reoptimizing the **base+pgcp** configuration) for all **pgcp** configurations. This methodology may slightly understate the benefits of profile-guided class prediction in the **base+IP$_G$+pgcp** configurations because any additional inlining enabled by interprocedural analysis would result in longer inlined chains of methods. Thus potentially more precise call-chain[1] profile information could be obtained by iteratively profiling the **base+IP$_G$+pgcp** configuration. However, this effect should be quite small and by using the same profile data for every configuration of a program, a variable is eliminated from the experiments.

All experiments were performed on a Sun Ultra 1 Model 170 SparcStation with 384 MB of physical memory and 2.3 GB of virtual memory that was running Solaris 5.5.1. For all programs/configurations, Vortex compiled the input programs to C code, which was then compiled using gcc version 2.7.2.1 with the -O2 option.

Each interprocedural configuration combines the impact of five interprocedural analyses. Appendix C contains the results of additional experiments that quantify the relative impact of each individual interprocedural analysis on bottom-line application performance.

4.1.3 Metrics

4.1.3.1 Call Graph Precision

A primary measure of call graph precision is the number of possibly invoked procedures at the various call sites in the call graph. Previous work has assessed the precision of context-sensitive call graph construction algorithms by reporting the average number of callee contours at a contour-level call site. This metric can be useful for comparing context-sensitivity schemes, but may not reflect the *effective precision* of the call graph. The key question is, can clients of the call graph (like the optimizer) exploit information computed at the contour granularity, or must information be summarized to a procedure granularity before it can be used? If only a single compiled copy of each procedure is produced, then contour-level precision metrics may be misleading, because

1. More information on call chain profiles, their interactions with inlining, and the details of profile-guided receiver class prediction in Vortex can be found elsewhere [Grove et al. 95].

during optimization all of a procedure's contours must be collapsed into a single summary view of the procedure. Some previous systems [Cooper et al. 92, Plevyak & Chien 95] have used the contours created during interprocedural analysis to drive procedure specialization, thus preserving any analysis-time specializations in the final compiled code. The Vortex compiler does not perform procedure specialization based on contours, therefore summarizing the contour-level information to a procedure granularity will more accurately reflect the effective call graph precision seen by later stages of Vortex. Tables A.1 and B.1 report the average number of callee procedures at a procedure-level call site and the percentage of all procedure-level call sites that have only one possible callee procedure. As part of its comparison of context-sensitive algorithms, table 4.3 reports the average number of callee contours at a contour-level call site.

4.1.3.2 Direct Costs of Call Graph Construction

The primary cost of a call graph construction algorithm is the compile time expended running the algorithm. Therefore, the primary metric used to assess the direct cost of a particular algorithm is the CPU time[2] consumed constructing the call graph. An additional concern is the amount of memory consumed during call graph construction; if the working set of a call graph construction algorithm does not fit into the available physical memory then, due to paging effects, CPU time will not accurately reflect the real (wall clock) time required for call graph construction. Algorithm memory requirements are approximated by measuring the growth in compiler heap size during call graph construction; this metric is somewhat conservative, because it does not account for details of the garbage collection algorithm and thus may overestimate an algorithm's peak memory requirements. An additional metric, the average number of contours per procedure, is reported for the context-sensitive call graph construction algorithms in section 4.4. The purpose of this metric is to approximately reflect the amount of analysis-time specialization performed by an algorithm.

---

2. Combined system and user mode CPU time as reported by **`rusage`**. There are three reasonable definitions of time: elapsed (wall clock) time, combined system and user CPU time, and user CPU time. The first metric is inappropriate because it includes time spent waiting for I/O events and even time in which the program was not actually running due other system activity (experiments were run on an otherwise unloaded machine, but it was not disconnected from the network or run in single-user mode). Arguments can be made for reporting just user-mode CPU time, but this metric excludes all OS operations performed on behalf of the program, in particular register window management on the SPARC. Since this can be a significant portion of program execution time (and on other architectures would manifest as time spent executing spill instructions in procedure prologues and epilogues), we chose to include system CPU time as part of total analysis time.

4.1.3.3 Impact on Application Performance

Call graph precision can affect both the time spent performing interprocedural analyses and the quality of the information computed by the interprocedural analyses, thus affecting bottom-line application performance. To reduce the amount of data presented, this chapter will mainly report on the bottom-line impact of call graph precision on application performance. Two metrics are used: application runtime (CPU time) and application code size.[3] These metrics are not perfect; in particular, instruction caching effects can cause application runtime to noticeably vary independent of any optimizations enabled by interprocedural analysis. For example, simply running the `strip` utility on an executable was observed to yield changes of +/- 8% in application runtime (`strip` simply removes the symbol table and relocation bits from the executable). Therefore, minor variations in application runtime should not be considered significant. Appendix B.2 contains additional experimental data that directly report on the effectiveness of optimizations enabled by particular analyses. The time taken to perform the MOD, escape, and exception detection analyses is reported in appendix B.1.

4.2 Potential Benefits of Interprocedural Analysis

This section establishes lower and upper bounds for the potential importance of call graph precision on the bottom-line impact of Vortex's current suite of interprocedural analyses on the programs in the benchmark suite. By providing an estimate of the total potential benefits of an extremely precise call graph and the fraction of those benefits achievable by very simple call graph construction algorithms, this section helps place the experimental assessment of the various other call graph construction algorithms in context.

3. Excluding fixed-size runtime system libraries.

4.2.1 Algorithm Descriptions

4.2.1.1 "Lower Bound" Algorithms

A true lower bound on the benefit of call graph precision could be obtained by performing interprocedural analysis using $G_{\perp}$, the call graph in which every call site is assumed to invoke every procedure in the program. However, $G_{\perp}$ is needlessly pessimistic; a much more precise call graph, $G_{selector}$, can be constructed with only superficial analysis of the program's source text. $G_{selector}$ improves $G_{\perp}$ by removing call edges between call sites and procedures with incompatible message names or numbers of arguments (Because Vortex's front ends use name mangling[4] to disambiguate any static overloading of message names, $G_{selector}$ will also take advantage of some static type information.)

Limited analysis of the program can be used to improve the precision of $G_{selector}$ along two axes. First, class hierarchy analysis (G_{CHA}) could be used to improve $G_{selector}$ by exploiting the information available in static type declarations (only methods that are applicable to classes that are subtypes of the receiver can be invoked) and specialized formal parameters (if within the body of a method a message is sent to one of the specialized formals of the enclosing method, then only methods that are applicable to the specializing class or its subclasses can be invoked). In statically typed languages with unified inheritance and type hiearchies, exploiting specialized formal parameters is just a special case of exploiting static type declarations, however in languages like Cecil or Smalltalk the two sources of information are separable. A number of systems have used class hierarchy analysis to resolve message sends and build program call graphs [Fernandez 95, Dean et al. 95b, Diwan et al. 96, Bacon & Sweeney 96, Bairagi et al. 97]. Second, rather than assuming that all methods declared in the program are invocable, the call graph construction algorithm could optimistically assume that a method is not reachable until it discovers that some reachable procedure instantiates a class to which the method is applicable ($G_{reachable}$). This optimistic computation of method and class liveness is the novel idea of Bacon and Sweeney's Rapid Type Analysis (G_{RTA}), a linear-time call graph construction algorithm that combines both

4. Name mangling extends function names to include an encoding of the static types of the arguments and/or return value. For example, a method foo(x:int,y:int):float might have a mangled name foo_I_I_rF.

class hierarchy analysis and optimistic reachability analysis to build a call graph [Bacon & Sweeney 96].

Finally, as in Diwan's Modula-3 optimizer [Diwan et al. 96], intraprocedural class analysis could be used to increase the precision of $G_{selector}$, G_{CHA}, $G_{reachable}$, or G_{RTA} by improving the analysis of any messages that are sent to objects that were created within the method.

4.2.1.2 "Upper Bound" Algorithms

The use of G_{ideal} as the basis for interprocedural analysis would provide an upper bound on the potential benefits of call graph precision for interprocedural analysis, since by definition G_{ideal} is the most precise sound call graph. Unfortunately, G_{ideal} is generally uncomputable. However, a loose upper bound can be established by using profile data to build some G_{prof_i}, an exact representation of the calls that occurred during one particular run of the program. G_{prof_i} is *not* a sound call graph, but does conservatively approximate the program's behavior on a particular input (all of our benchmarks are deterministic). To obtain an upper bound on the performance benefits of call graph precision, the optimizer is instructed to assume that G_{prof_i} *is* sound and the resulting optimized program is rerun on the *same* input. Note that if the IP- G_{prof_i} optimized program was run on any other input, it could very well compute an incorrect result.

In theory, G_{prof_i} will be at least as precise as G_{ideal}. However, limitations in Vortex's profiling infrastructure[5] prevent us from gathering fully context-sensitive profile-derived class distributions and thus the actual profile-derived call graph G_{prof} that Vortex builds is less precise than G_{prof_i}. It is, however, at least as precise as the optimal context-insensitive call graph. In the other direction, G_{prof_i} may also be too loose of an upper bound if the single run does not fully exercise the program. I believe that this second concern (too loose of an upper bound) is unlikely to be a significant problem for these benchmark programs. The input data sets I chose should exercise most portions of the program, and most of the programs are compiler-like applications whose behavior is not highly-dependent on the details of their input. To partially support this claim, I built

5. Vortex's profiling infrastructure is described in much more detail elsewhere [Grove et al. 95]. The key issue is that the profile-derived class distributions gathered by Vortex are tagged with finite-length segments of the dynamic call chain, limited by the degree of method inlining. Building a context-sensitive G_{prof_i} may require profile data to be tagged with arbitrarily long finite segments of the dynamic call chain.

a second G_{prof} configuration for the compiler program that used a less optimistic call graph constructed by combining the profile data gathered from six different runs of the program: some on a single file, some on 30 files, at varying levels of optimization. This combined G_{prof} was only 2% slower than the single-run G_{prof}.

4.2.2 Experimental Assessment

To establish lower and upper bounds on the bottom-line impact of call graph precision, interprocedural analysis was performed using two of the call graph construction algorithms discussed above: $G_{selector}$ and G_{prof}. Figure 4.1 displays normalized execution speeds for the **base**, **base+IP$_{selector}$**, **base+IP$_{prof}$**, **base+pgcp**, **base+IP$_{selector}$+pgcp**, and **base+IP$_{prof}$+pgcp** configurations of each application. Throughout this chapter we will use the convention of stacking the bars for the two configurations that differ only in the presence/absence of profile-guided class prediction on top of each other, with the static bar shaded and the additional speedups enabled by profile-guided class prediction indicated by the white bar. The absence of the additional white bar indicates that profile-guided class prediction had no impact on performance; the case in which *x*+pgcp is measurable slower than *x* will be indicated by an explicit comment in the running text. To emphasize that **base** is already highly optimized, an **unopt** configuration is shown as well in which no Vortex optimizations were performed (but the resulting C files were still compiled by gcc -O2); the difference between the **unopt** and **base** configurations indicates the effectiveness of Vortex's basic optimizations suite. The difference between the **base** and **IP$_{prof}$** configurations approximates the maximum speedup achievable in the current Vortex system by an arbitrarily precise call graph construction algorithm. The **IP$_{selector}$** configurations illustrate how much of this benefit can be obtained by an extremely simple and cheap call graph construction algorithm. More ambitious call graph construction algorithms may be profitable if there is a substantial performance gap between the **IP$_{selector}$** and **IP$_{prof}$**. configurations.

For almost all the programs, the performance difference between **base+IP$_{selector}$** and **base+IP$_{prof}$** was almost as large as that between **base** and **base+IP$_{prof}$**, indicating that more aggressive call graph algorithms than $G_{selector}$ will be required to reap the potential benefits of interprocedural analysis. For the Cecil programs, the potential benefit is quite large. **base+IP$_{prof}$** dominates **base+prof** and on the four larger benchmarks yields speedups ranging from a factor of

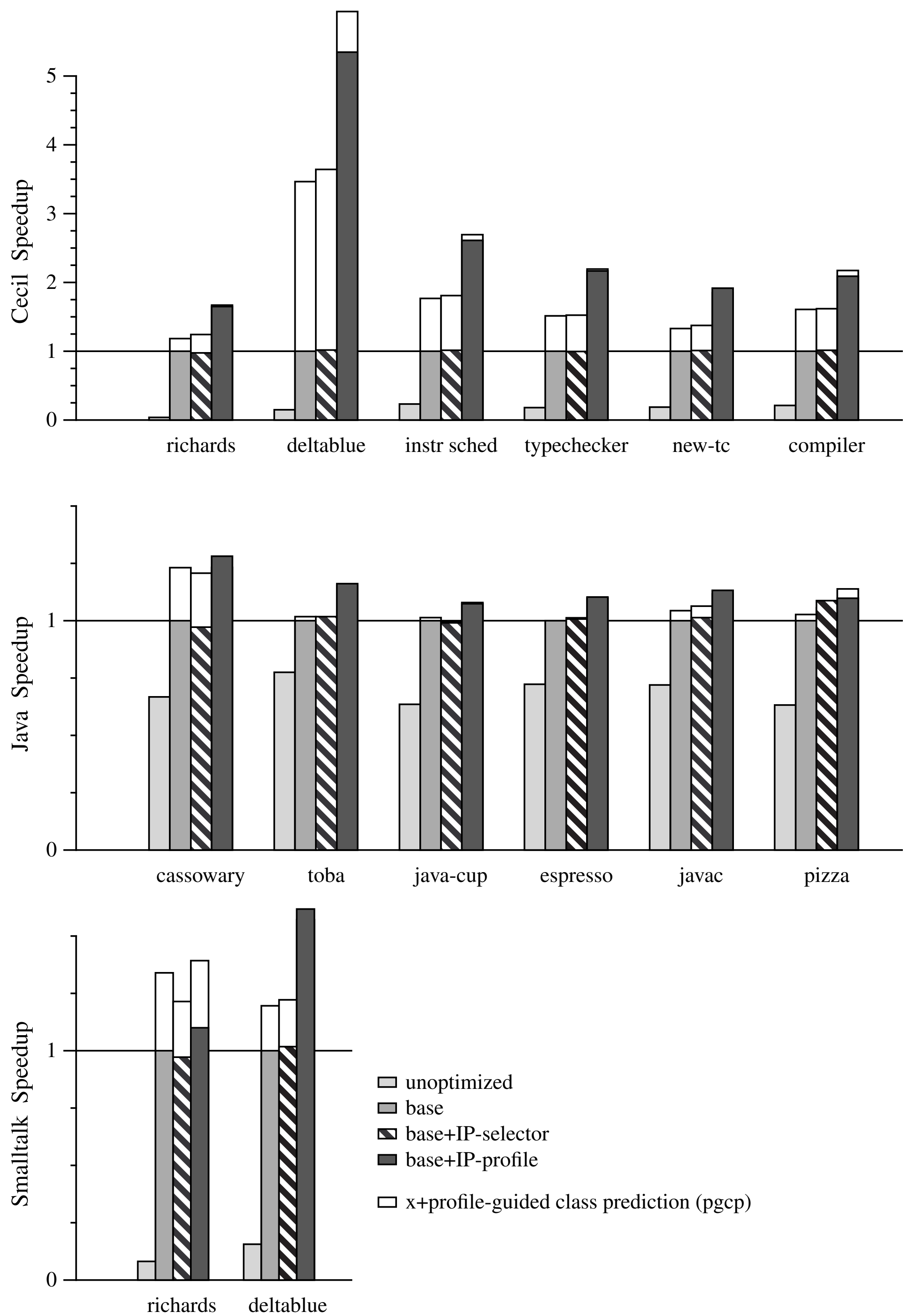


Figure 4.1: Upper and lower bounds on the impact of call graph precision

1.9 to 2.6 over **base**. The potential improvements for the Java benchmarks are much smaller; this is not entirely unexpected because the combination of Java's hybrid object model and its static type system both make **unopt** more efficient than it is in Cecil or Smalltalk and make the optimizations included in the **base** configuration more effective, thus leaving even less remaining overhead for interprocedural analysis to attack. For cassowary, **base+IP$_{\textbf{prof}}$** is roughly 25% faster than **base**, and the potential benefits are smaller for the remainder of the Java programs.

4.3 The Basic Algorithm: 0-CFA

The 0-CFA algorithm is the classic context-insensitive, data flow-sensitive call graph construction algorithm. It produces a more precise call graph than $G_{selector}$ and the other "lower bound" algorithms of section 4.2.1.1 by performing an iterative interprocedural data and control flow analysis of the program during call graph construction. The resulting more precise call graph may enable substantial improvements over those provided by $G_{selector}$. The 0-CFA algorithm was first described in the context of control flow analysis for Scheme programs by Shivers [Shivers 88, Shivers 91]. The basic 0-CFA strategy has been used in a number of call graph construction algorithms; Palsberg and Schwartzbach's basic algorithm [Palsberg & Schwartzbach 91], Hall and Kennedy's call graph construction algorithm for Fortran [Hall & Kennedy 92], and Lakhotia's algorithm for building a call graph in languages with higher-order functions [Lakhotia 93] are all derived from 0-CFA.

4.3.1 Algorithm Description

0-CFA uses the single-point lattice (the lattice with only a $\perp$ value) for its *ProcKey*, *InstVarKey*, and *ClassKey* partial orders. The general algorithm is instantiated to 0-CFA by the following parameters:

$PKS(caller : \text{ProcContour}, c : \text{CallSite}, a : \text{AllTuples(ClassContourSet)}, callee : \text{Procedure}) \rightarrow \{\perp\}$
$IVKS(i : \text{InstVariable}, b : \text{ClassContourSet}) \rightarrow \{\perp\}$
$CKS(c : \text{Class}, p : \text{ProcContour}) \rightarrow \{\perp\}$
$EKS(c : \text{Closure}, p : \text{ProcContour}) \rightarrow \{\perp\}$
$CIF : p = \infty$
$SIF : \emptyset$

For every procedure, instance variable, and class, the 0-CFA contour key selection functions return the singleton set containing the only element of the matching contour key partial order, leading to the selection of the single contour that is the only analysis-time representation of the procedure, instance variable, or class. By uniformly setting $p = \infty$, 0-CFA always generates simple inclusion constraints and all non-specified class contour sets are optimistically initialized to the empty set.

4.3.2 Experimental Assessment

Figure 4.2 reports the costs of using the 0-CFA algorithm by plotting call graph construction time and heap space growth as a function of program size, measured by the number of call sites in the program. Note that both y-axis use a log scale. For the three largest Cecil programs, call graph construction was a substantial cost: consuming roughly 10 minutes of CPU time for typechecker and new-tc and just over three hours for compiler. These large analysis times are not due to paging effects. Although heap growth during call graph construction was substantial, 60 MB for typechecker and new-tc and 200 MB for compiler, there were 384 MB of physical memory on the machine and CPU utilization averaged over 95% during call graph construction. The 0-CFA algorithm exhibited much better scalability on the Java programs: even for the largest programs call graph construction only consumed a few minutes of CPU time. Although the largest Java program is significantly smaller than the largest Cecil program (23,000 call sites vs. 38,000), this may not explain the apparent qualitative difference in the 0-CFA call graph construction times. To reliably compare the scalability of the 0-CFA algorithm in Cecil and Java, a few larger Java programs (40,000+ call sites) would be required. For comparably sized programs, it appears that 0-CFA call graph construction takes roughly a factor of four longer for Cecil than for Java. I suspect that a major cause of this difference is the heavy use of closures in the Cecil programs, which results in a much larger number of analysis time "classes" in Cecil programs, and thus increases the amount of propagation work. For example, typechecker contains 635 classes and 3,900 closures for an effective total of 4,535 classes, while pizza contains only 335 class and no closures (Table A.1 reports the number of classes and closures in each benchmark program). For the two Smalltalk programs, call graph construction took approximately 25 minutes and 40 MB of heap space.

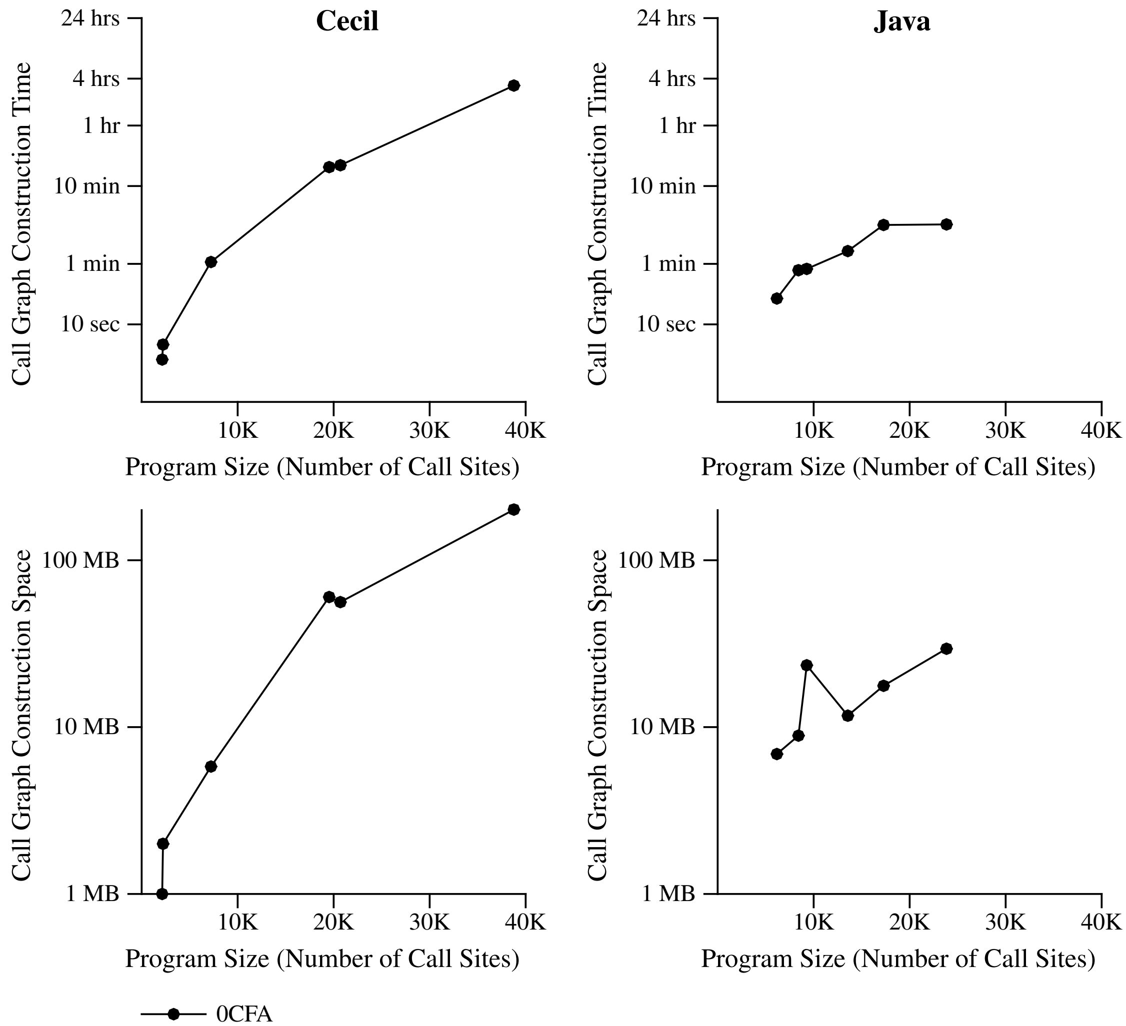


Figure 4.2: Costs of 0-CFA call graph construction

Figure 4.3 reports the bottom-line performance impact of 0-CFA by displaying the execution speeds of the two **IP$_{\textbf{0-CFA}}$** configurations of each benchmark (the third pair of bars). To enable easy comparison, the execution speeds obtained by the **base**, **IP$_{\textbf{selector}}$**, and **IP$_{\textbf{prof}}$** configurations are repeated. For the two smallest Cecil programs, 0-CFA enables almost all of the potentially available speedup embodied in the **IP$_{\textbf{prof}}$** configuration. In the four larger Cecil programs, 0-CFA enables roughly half of the potentially available speedup: it substantially out-performs **base** but the remaining large gap between it and **IP$_{\textbf{prof}}$** indicates that context-sensitivity might enable additional performance gains. The critical difference between the small and large Cecil programs

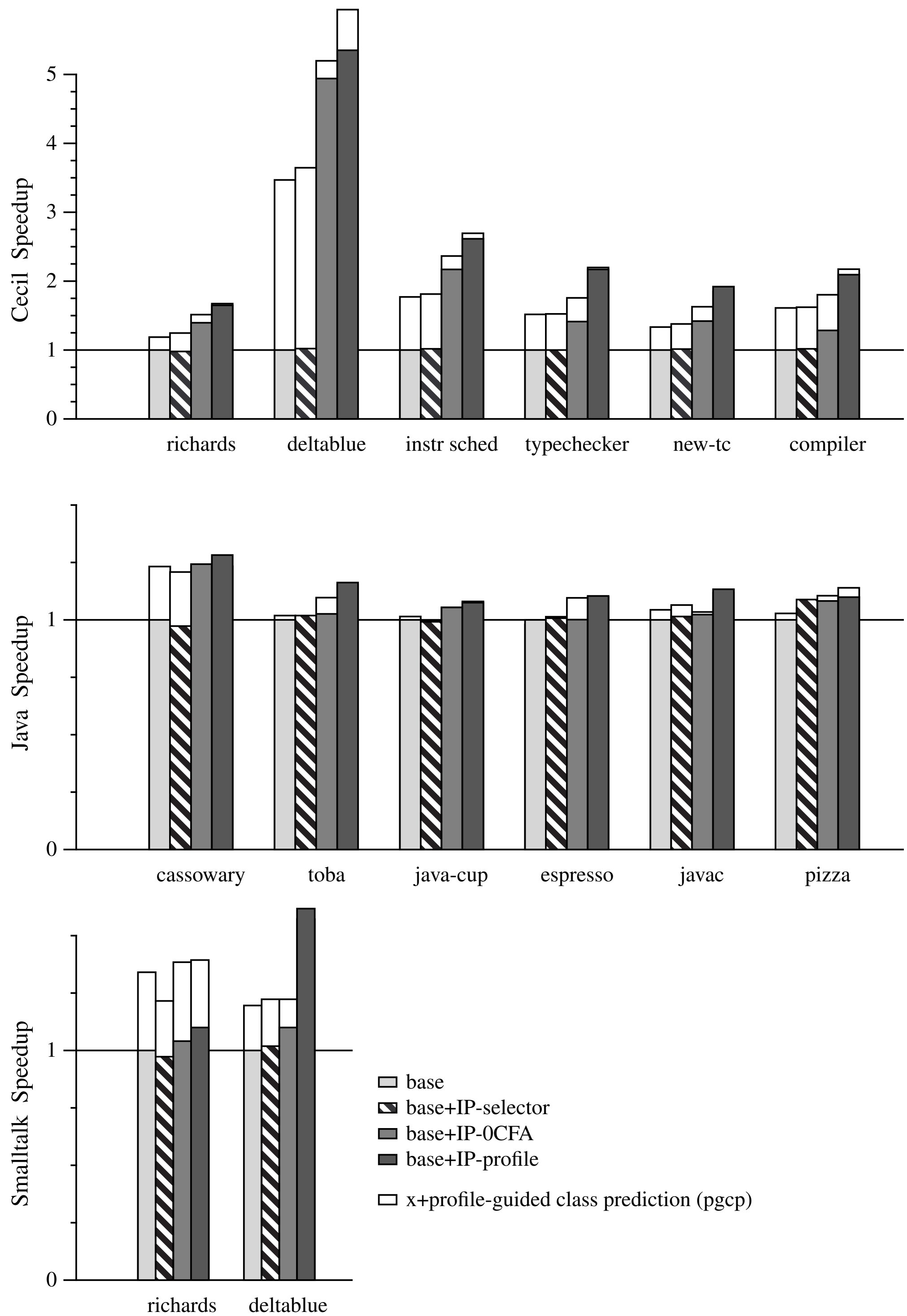


Figure 4.3: Performance impact of 0-CFA

is the amount of truly polymorphic code. Although all six programs use the same generic hash table, set, etc. library classes, richards and deltablue are too small to contain multiple clients that use these polymorphic library routines in different ways, i.e. to store objects of different classes. As the programs become larger, 0-CFA's inability to accurately analyze polymorphic code becomes more and more important. The performance impact of 0-CFA on Java programs was fairly bi-modal: for some programs it enables speedups comparable to those of $\mathbf{IP_{prof}}$, and for others it is ineffective.

4.4 Context-Sensitive Call Graph Construction Algorithms

In contrast to the 0-CFA algorithm, context-sensitive instances of the general algorithm will create more than one contour for some (or all) of a program's procedures, instance variables, and classes. This section defines and assesses three context-sensitive algorithms: *k-l*-CFA, CPA, and SCS. All of the algorithms described in this section generate only simple inclusion constraints, $\supseteq_\infty$.

4.4.1 Algorithm Descriptions

4.4.1.1 Context-Sensitivity based on Call Chains

One of the most commonly used forms of context-sensitivity is to use a vector of the *k* enclosing calling procedures at a call site to select the target contour for the callee procedure (the "call-strings" approach of Sharir and Pnueli [Sharir & Pnueli 81]). If $k = 0$, then this degenerates to the single-point lattice and a context-insensitive algorithm (0-CFA); $k = 1$ for *ProcKey* corresponds to analyzing a callee contour separately for each source-level call site, and $k = 1$ for *ClassKey* corresponds to treating each distinct source-level instantiation site of a class as giving rise to a separate class contour. An algorithm may use a fixed value of *k* throughout the program, as in Shivers's *k*-CFA family of algorithms for Scheme [Shivers 88, Shivers 91], Oxhøj's 1-CFA extension to Palsberg and Schwartzbach's algorithm [Oxhøj et al. 92] or various other adaptations of *k*-CFA to object-oriented programs [Vitek et al. 92, Phillips & Shepard 94]. More sophisticated adaptive algorithms try to use different levels of *k* in different regions of the call graph to more flexibly manage the trade-off between analysis time and precision [Plevyak & Chien 94, Plevyak 96]. Finally, a number of algorithms based on arbitrarily large finite values for *k* have been proposed: Ryder's call graph construction algorithm for Fortran 77 [Ryder 79], Callahan's extension to Ryder's work to support recursion [Callahan et al. 90], and Emami's alias analysis algorithm for C [Emami et al. 94] all treat each non-recursive path through the call graph as creating a new context. Alt and Martin have developed an even more aggressive call graph construction algorithm, used in their PAG system, that first "unrolls" k levels of recursion [Alt & Martin 95]. Steensgaard developed an unbounded-call-chain algorithm that handles nested lexical environments by applying a widening operation to class sets of formal parameters at entries to recursive cycles in the call graph [Steensgaard 94].

For object-oriented programs, Shivers's k-CFA family of algorithms can be straightforwardly extended to the k-l-CFA family of algorithms where k denotes the degree of context-sensitivity in the *ProcKey* domain and l denotes the degree of context-sensitivity in the *ClassKey* domain [Vitek et al. 92, Phillips & Shepard 94]. The partial orders used as contour keys by these algorithms are detailed below:

Algorithm	***ProcKey***	***InstVarKey***	***ClassKey***
k-0-CFA where k>0	*AllTuples(Procedure)*	single-point lattice	single-point lattice
k-l-CFA where $k \geq l > 0$	*AllTuples(Procedure)*	*ClassContour*	*AllTuples(Procedure)*

The contour key selection functions are defined using two auxiliary function $\oplus$ and F_w where $x \oplus \langle y_1, \ldots, y_k \rangle = \langle x, y_1, \ldots, y_k \rangle$ and $F_w(x, \langle y_1, \ldots, y_w, \ldots, y_k \rangle) = \langle x, y_1, \ldots, y_{w-1} \rangle$. The general algorithm is instantiated with the following parameters for the k-l-CFA family of algorithms:

$$PKS(caller : \text{ProcContour}, c : CS, a : \text{AT(CCS)}, callee : \text{P}) \rightarrow \{F_k(Proc(ID(caller)), Key(ID(caller)))\}$$

$$IVKS(i : \text{InstVariable}, b : \text{ClassContourSet}) \rightarrow \begin{cases} \{\bot\}, l = 0 \\ b, \text{otherwise} \end{cases}$$

$$CKS(c : \text{Class}, p : \text{ProcContour}) \rightarrow \begin{cases} \{\bot\}, l = 0 \\ \{\langle F_l(Proc(ID(p)), Key(ID(p)))\rangle\}, \text{otherwise} \end{cases}$$

$$EKS(c : \text{Closure}, p : \text{ProcContour}) \rightarrow \begin{cases} \{\bot\}, \text{if } c \text{ contains no non-global free variables} \\ \{Key(ID(p)) \oplus lc \mid lc \in Lex(ID(p))\}, \text{otherwise} \end{cases}$$

$$CIF : p = \infty$$
$$SIF : \emptyset$$

Thus, the procedure context-sensitivity strategy used in the k-l-CFA family of algorithms is identical to that used in the original k-CFA algorithms. If $l > 0$, the *IVKS* and *CKS* functions collaborate to enable the context-sensitive analysis of instance variables. *CKS* tags classes with the contour in which they were created. A separate class contour set (representing the contents of an instance variable) is maintained for each *Class* and *ClassKey* pair. IVKS uses the *ClassContourSet* of the base expression of the instance variable load or store to determine which instance variable contours should be used to analyze the access. Note that in practice k will be greater than or equal to l because *ClassKeys* are based on *ProcKeys*, and larger values of l will not have any additional benefit.

4.4.1.2 Context-Sensitivity based on Parameters

Another commonly used basis for context-sensitive analysis of procedures is some abstraction of the actual parameter values that are passed to the procedure from its call sites. For example, an abstraction of the alias relationships among actual parameters has been used as the basis for context-sensitivity in algorithms for interprocedural alias analysis [Landi & Ryder 91, Wilson & Lam 95]. Similarly, several call graph construction algorithms for object-oriented languages use information about the classes of actual parameters as the critical input to their procedure contour key selection functions. These algorithms attempt to improve on the brute-force approach of call-chain based context-sensitivity by using more sophisticated notions of which callers are similar, and thus can share the same callee contour, and which callers are different enough to require distinct callee contours.

Two such algorithms are the Cartesian Product Algorithm (CPA) [Agesen 95] and the Simple Class Set algorithm (SCS) [Grove et al. 97]. The primary difference in the algorithms is their procedure contour key selection function. CPA uses $AllTuples(ClassContour)$ as its ProcKey partial order, but SCS uses $AllTuples(ClassContourSet)$. Both algorithms use the single-point lattice for their *InstVarKey* and *ClassKey* partial orders. The general algorithm is instantiated with the following parameters for CPA:[6]

$PKS(caller : \text{ProcContour}, c : CS, a : \text{AllTuples(ClassContourSet)}, callee : \text{Proc}) \rightarrow CPA(a)$
where $CPA(\langle S_1, S_2, \ldots, S_n \rangle) = S_1 \times S_2 \times \ldots \times S_n$
$IVKS(i : \text{InstVariable}, b : \text{ClassContourSet}) \rightarrow \{\bot\}$
$CKS(c : \text{Class}, p : \text{ProcContour}) \rightarrow \{\bot\}$
$EKS(c : \text{Closure}, p : \text{ProcContour}) \rightarrow \begin{cases} \{\bot\}, \text{ if } c \text{ contains no non-global free variables} \\ \{Key(ID(p)) \oplus lc \mid lc \in Lex(ID(p))\}, \text{ otherwise} \end{cases}$
$CIF : p = \infty$
$SIF : \varnothing$

and SCS uses:

$PKS(caller : \text{ProcContour}, c : CS, a : \text{AllTuples(ClassContourSet)}, callee : \text{Proc}) \rightarrow \{a\}$
$IVKS(i : \text{InstVariable}, b : \text{ClassContourSet}) \rightarrow \{\bot\}$
$CKS(c : \text{Class}, p : \text{ProcContour}) \rightarrow \{\bot\}$
$EKS(c : \text{Closure}, p : \text{ProcContour}) \rightarrow \begin{cases} \{\bot\}, \text{ if } c \text{ contains no non-global free variables} \\ \{Key(ID(p)) \oplus lc \mid lc \in Lex(ID(p))\}, \text{ otherwise} \end{cases}$
$CIF : p = \infty$
$SIF : \varnothing$

Both algorithms generate inclusion constraints ($p = \infty$) and by default initialize class contour sets to the empty set. In CPA, for each callee procedure the procedure contour key selection function computes the cartesian product of the actual parameter class contour sets and a procedure contour is selected/created for each element. In SCS, for each callee procedure the procedure contour key selection function returns a single procedure contour key, which is exactly the tuple of actual parameter class contour sets a, and thus only a single contour per callee procedure is created/selected to analyze the call site.

To obtain contour reuse, CPA breaks the analysis of a callee procedure into a number of small pieces in the hope that there will be other call sites of the procedure with overlapping tuples of class contour sets that will be able to reuse some of the contours created and analyzed to handle the first call site. In contrast, SCS can only reuse a callee contour if there are two call sites of a procedure that have identical tuples of class contour sets passed as actual parameters. On the other

6. Agesen's implementation of CPA for Self actually uses a more complex *EKS* function that further reduces the number of closure contours by "fusing" closure contours that reference the same subset of its enclosing procedure's formal parameters. This optimization is difficult to express in our framework and is not included in the Vortex implementation of CPA. See section 4.4 of [Agesen 96] for more detail.

hand, CPA may needlessly create a large number of contours to analyze a callee that will not benefit from the more precise formal class contour sets.

To illustrate the context-sensitivity strategies of CPA and SCS, figure 4.4 contains an example program and the resulting SCS and CPA call graphs. Each node in the call graph represents a contour and is labeled with the *Procedure* and *ProcKey* components of its *ProcID*. If multiple contours were created for a procedure they are grouped by a dashed oval and labeled with the method name. Notice that in the SCS call graph at most one contour per procedure can be invoked from each call site. This example also illustrates one way in which CPA may produce a more precise call graph than SCS. In the CPA call graph, the contours representing **`double(@num)`** only invoke contours representing **`+(@int,@int)`** and **`+(@float,@float)`**. However in the SCS call graph, **`double(@num)`** also invokes a contour for **`+(@num,@num)`**.

In the worst case, CPA may require $O(N^a)$ contours to analyze a call site, where a is the number of arguments at the call site. Under the assumption that a is bounded by a constant, the worst case analysis time of CPA will be polynomial in the size of the program [Agesen 96]. In the worst case, SCS may require $O\left(N^{N^a}\right)$ contours to analyze a program. To avoid requiring an unreasonably large number of contours in practice, Agesen actually implements a variant of CPA that we term bounded-CPA (or b-CPA) that uses a single context-insensitive contour to analyze any call site at which the number of terms in the cartesian product of the actual class sets exceeds a threshold value. Similarly, a bounded variant of SCS, b-SCS, can be defined to limit the number of contours created per procedure by falling back on a context-insensitive summary when the procedure's contour creation budget is exceeded. The experimental results in section 4.4.2 only present data for the b-CPA and SCS algorithms. Unbounded CPA does not scale to large Cecil programs [Grove et al. 97]. Bounded SCS is not considered because for the majority of our benchmark programs unbounded SCS actually required less analysis time than b-CPA despite its worst-case exponential time complexity.

In the presence of lexically nested functions, both CPA and SCS are vulnerable to recursive customization, because the class contour key selection function for the closure class must encode the lexically enclosing contour. This leads to a mutual recursion between the *ProcKey* and *ClassKey* partial orders which results in an infinitely tall call graph domain. Agesen defines the recursive customization problem and gives three methods for conservatively detecting when it

```
class num;                      procedure A() {
   method +(@num,@num){            if (random()) {
      ....                            x := 3; y := 5;
   }                               } else {
   method double (x@num) {            x := 3.5; y := 4.5;
      return x + x;                }
   }                               return x + double(y);
                                }
class float inherits num;       procedure B() {
   method +(@float,@float){        if (random()) {
      ....                            x := 3; y := 5;
   }                               } else {
                                      x := 3.5; y := 5;
class int inherits num;            }
   method +(@int,@int){            return x + double(y);
      ....                      }
   }
```

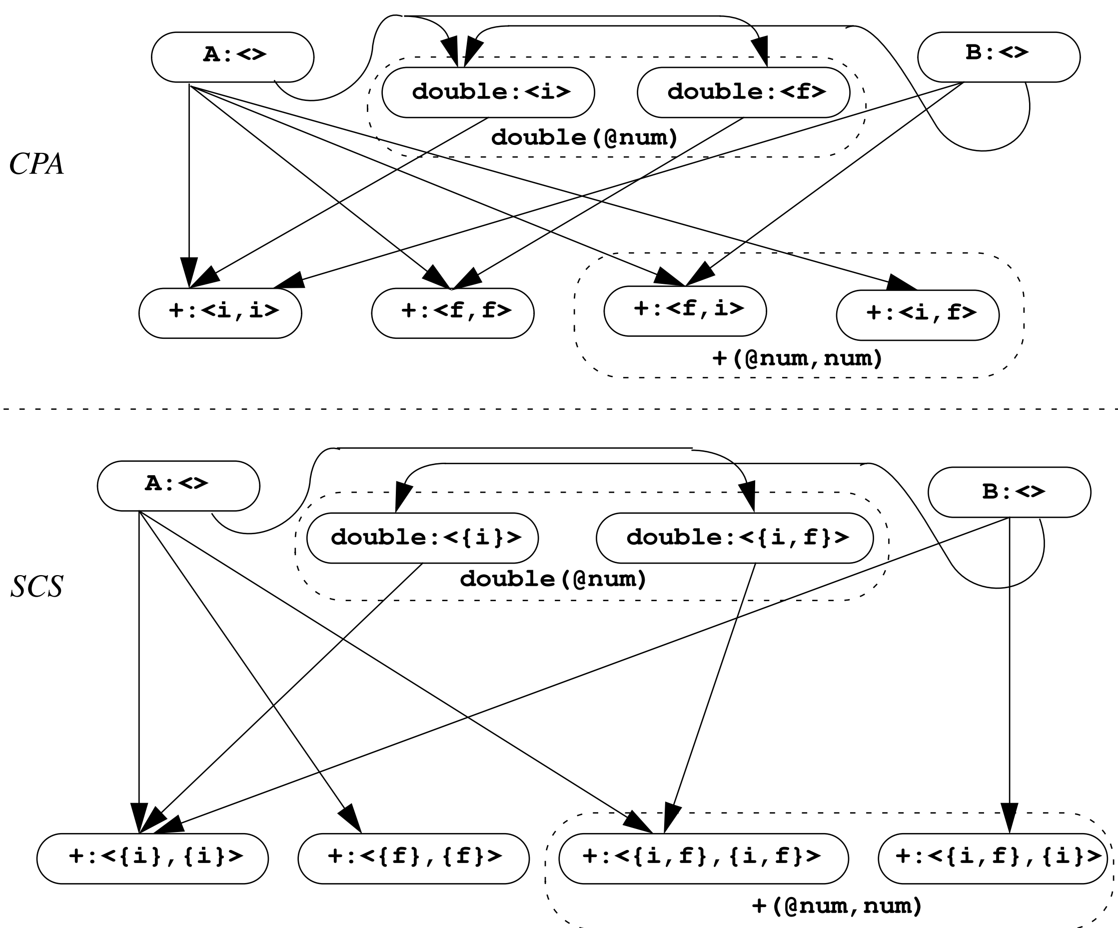


Figure 4.4: CPA and SCS example

occurs (thus enabling a widening operation to be applied in the procedure contour key selection function) [Agesen 96]. The Vortex implementations of CPA and SCS use the weakest of these three methods: programmer annotation of methods that may induce recursive customization.

Other context-sensitive call graph construction algorithms for object-oriented languages have been defined by exploiting similar ideas. The "eager splitting" used as a component of each phase of Plevyak's iterative refinement algorithm [Plevyak 96] is equivalent to unbounded CPA; thus I expect that an implementation of Plevyak's algorithm would also not scale to large Cecil programs. Pande's algorithm for interprocedural class analysis in C++ [Pande & Ryder 94] is built upon Landi's alias analysis for C [Landi & Ryder 91] and uses an extension of Landi's conditional points-to information as the basis for its context-sensitivity. Prior to developing CPA, Agesen proposed the hash algorithm to improve the analysis of Self programs [Agesen et al. 93]. The hash algorithm makes context-sensitivity decisions by hashing a description of the calling context; call sites of a target method that compute the same hash value will share the same callee contour. The original hash algorithm computed a simple hash function from limited information that returned a single value (thus restricting the analysis to use only a single callee contour per call site), but Agesen later extended the hash algorithm to allow the hash function to include the current sets of argument classes as part of its input and to return multiple hash values for a single call site [Agesen 96].

4.4.2 Experimental Assessment

The time and space costs of a subset of the context-sensitive call graph construction algorithms are plotted as functions of program size in figure 4.5; for comparison the costs of the 0-CFA algorithm are also included. For many of the program and algorithm combinations, call graph construction did not complete in less than 24 hours of CPU time. In particular, none of the context-sensitive algorithms could successfully analyze compiler, and only 1-0-CFA completed on typechecker and new-tc. All of the context-sensitive algorithms failed on both Smalltalk programs (1-0-CFA ran for over 72 CPU hours without terminating on richards), and higher values of *k*-*l*-CFA failed to complete on the three largest Java programs. For the Java and small Cecil programs, the more intelligent context-sensitivity strategies of CPA and SCS resulted in lower analysis-time costs than the brute force approach of *k*-*l*-CFA.

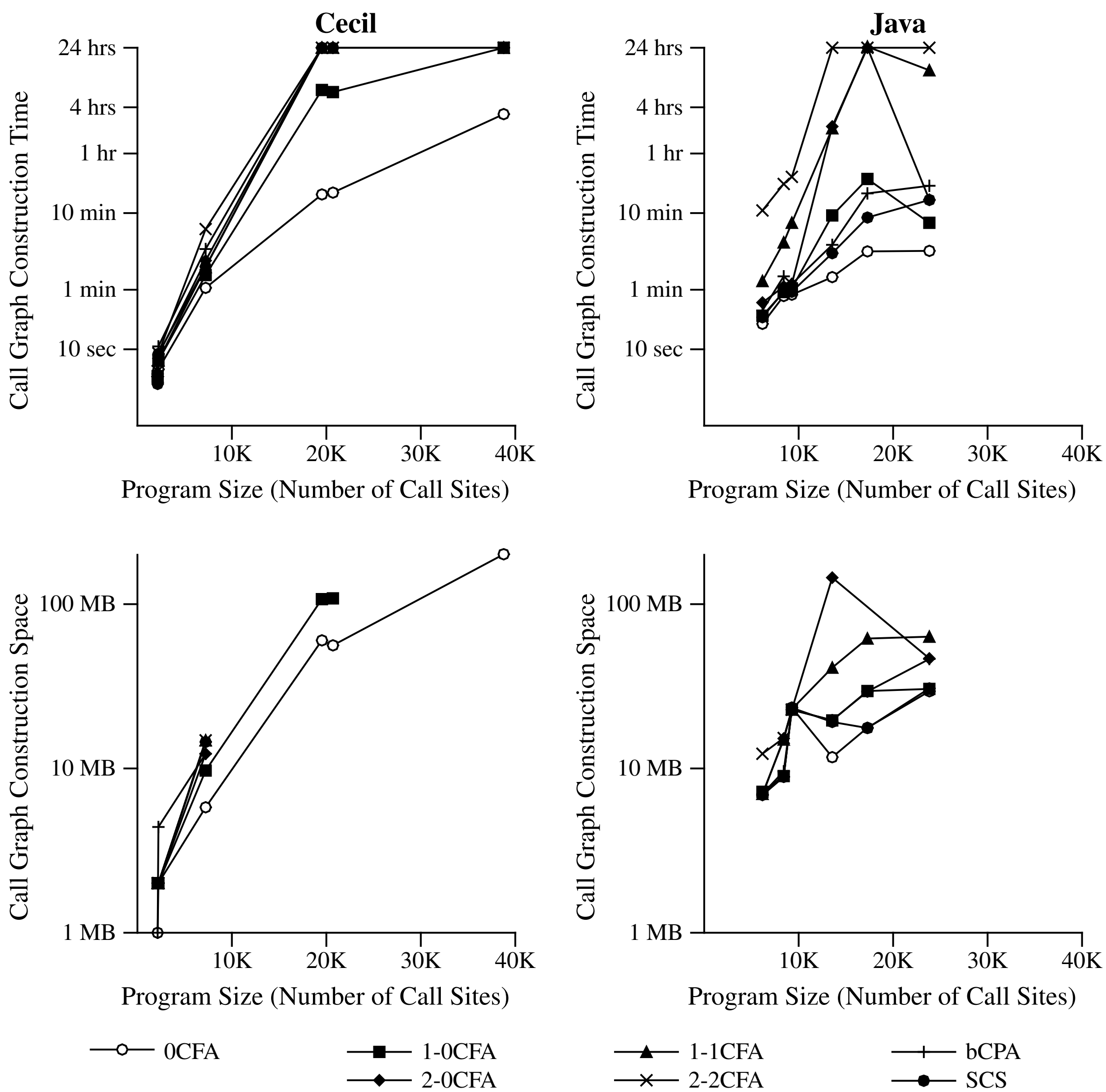


Figure 4.5: Costs of context-sensitive call graph construction

Table 4.3 reports three additional metrics that quantify the degree of analysis-time specialization and contour reuse.

- *Contours per Node*: the average number of contours created for each procedure. This number is suggestive of the extent of the analysis-time specialization performed by the algorithm.

Table 4.3: Contour Statistics[a]

	Program	Metric	SCS	b-CPA	1-0-CFA	2-0-CFA	3-0-CFA	1-1-CFA	2-2-CFA	3-3-CFA
Cecil	richards	Contours per Node	2.12	2.75	1.93	2.68	3.16	1.93	2.68	3.16
		Contours per CallSite	1.12	1.32	1.08	1.07	1.08	1.08	1.07	1.08
		CallSites per Contour	1.58	1.87	1.57	1.24	1.19	1.19	1.24	1.19
	deltablue	Contours per Node	3.98	5.00	2.49	3.79	5.33	2.45	3.73	5.22
		Contours per CallSite	1.21	1.49	1.25	1.20	1.18	1.23	1.19	1.17
		CallSites per Contour	1.96	2.10	1.85	1.56	1.49	1.49	1.56	1.48
	instr sched	Contours per Node	6.72	8.88	3.30	6.65	13.02	3.28	6.56	12.49
		Contours per CallSite	1.43	1.63	1.48	1.41	1.40	1.48	1.40	1.40
		CallSites per Contour	2.41	2.64	2.85	2.43	2.41	2.41	2.40	2.38
	typechecker	Contours per Node			6.94					
		Contours per CallSite			5.45					
		CallSites per Contour			13.83					
	new-tc	Contours per Node			6.62					
		Contours per CallSite			5.32					
		CallSites per Contour			13.69					
	compiler	Contours per Node								
		Contours per CallSite								
		CallSites per Contour								
Java	cassowary	Contours per Node	1.71	2.16	2.31	4.99	11.30	2.16	4.32	9.31
		Contours per CallSite	1.14	1.56	1.05	1.06	1.06	1.03	1.04	1.05
		CallSites per Contour	4.84	6.76	3.14	2.95	2.47	2.47	2.89	2.41
	toba	Contours per Node	1.44	1.75	2.59	5.45	11.35	2.54	5.12	10.20
		Contours per CallSite	1.20	1.28	1.04	1.05	1.05	1.01	1.00	1.00
		CallSites per Contour	8.26	13.64	4.46	3.38	2.57	2.57	3.29	2.51
	java-cup	Contours per Node	1.60	1.67	2.68	5.18	10.18	2.56	4.98	9.37
		Contours per CallSite	1.18	1.29	1.03	1.04	1.04	1.02	1.02	1.02
		CallSites per Contour	6.37	8.14	3.69	3.19	2.51	2.51	3.08	2.42
	espresso	Contours per Node	2.21	2.41	4.67	48.78		4.52		
		Contours per CallSite	2.27	3.21	3.01	4.18		2.87		
		CallSites per Contour	14.36	20.20	18.61	31.72		17.95		
	javac	Contours per Node	4.38	4.66	5.57			5.55		
		Contours per CallSite	2.42	3.90	3.73			3.72		
		CallSites per Contour	.92	20.58	17.82			17.86		
	pizza	Contours per Node	4.42	5.69	3.69	12.01	38.40	3.66		
		Contours per CallSite	1.24	2.08	1.13	1.16	1.18	1.12		
		CallSites per Contour	14.76	18.14	6.15	5.94	5.76	6.11		

a. Shaded cells correspond to configurations that either did not complete in 24 CPU hours or exhausted available virtual memory (2.3 GB).

- *Contour per CallSite*: the average number of contours created to analyze a contour-level call site. This number can indicate two things. For *k-l*-CFA, which uses exactly one contour for each callee procedure, it indicates the average number of possibly invoked callee procedures at each contour call site. CPA may use multiple contours per procedure to analyze a call site, so this number simply indicates on average how many contours are used at each contour call site. At any given moment, SCS is only using one contour per callee procedure to analyze a call site, but as the class sets of the call's actual parameters grow it will create new contours and discard old contours. Thus, for SCS, this number indicates on average how many different contours were used at some point during the analysis to analyze a contour call site.

- *CallSites per Contour*: the average number of contour call sites that invoke a given contour. This metric suggests the degree of contour reuse; larger numbers imply more sharing of callee contours across call sites.

The most interesting aspect of this data is the comparison of the CPA and SCS contour reuse strategies it enables. For all the programs, both algorithms are reusing contours across call sites, but b-CPA gets substantially more reuse than SCS due to its finer-grained creation of contours. However, for all the programs, on average b-CPA creates more contours per procedure than SCS, thus incurring more analysis time work. This agrees with the analysis time data of figure 4.5 in which SCS call graph construction is usually faster than b-CPA.

Finally, figure 4.6 shows the bottom-line benefits for our benchmark suite of the additional call graph precision enabled by context-sensitive interprocedural class analysis; for comparison the speedups obtained by $\mathbf{IP_{0\text{-}CFA}}$ and $\mathbf{IP_{prof}}$ are also shown. The bars are grouped into three sets, the **base**, $\mathbf{IP_{selector}}$, and $\mathbf{IP_{0\text{-}CFA}}$ configurations, the $\mathbf{IP}_{\boldsymbol{k\text{-}l}\textbf{-CFA}}$ configurations, and the $\mathbf{IP_{b\text{-}CPA}}$, $\mathbf{IP_{SCS}}$, and $\mathbf{IP_{prof}}$ configurations. Missing bars indicate combinations that did not complete; since context-sensitive analysis of the Smalltalk programs was completely unsuccessful, the graph is omitted. Overall, despite some improvements in call graph precision (see table B.1), none of the context-sensitive analysis enabled a reliably measurable improvement over 0-CFA on the three smallest Cecil programs or on the Java programs. For typechecker and new-tc, 1-0CFA did improve over 0-CFA, but did not reach the performance of the $\mathbf{IP_{prof}}$ configurations, suggesting

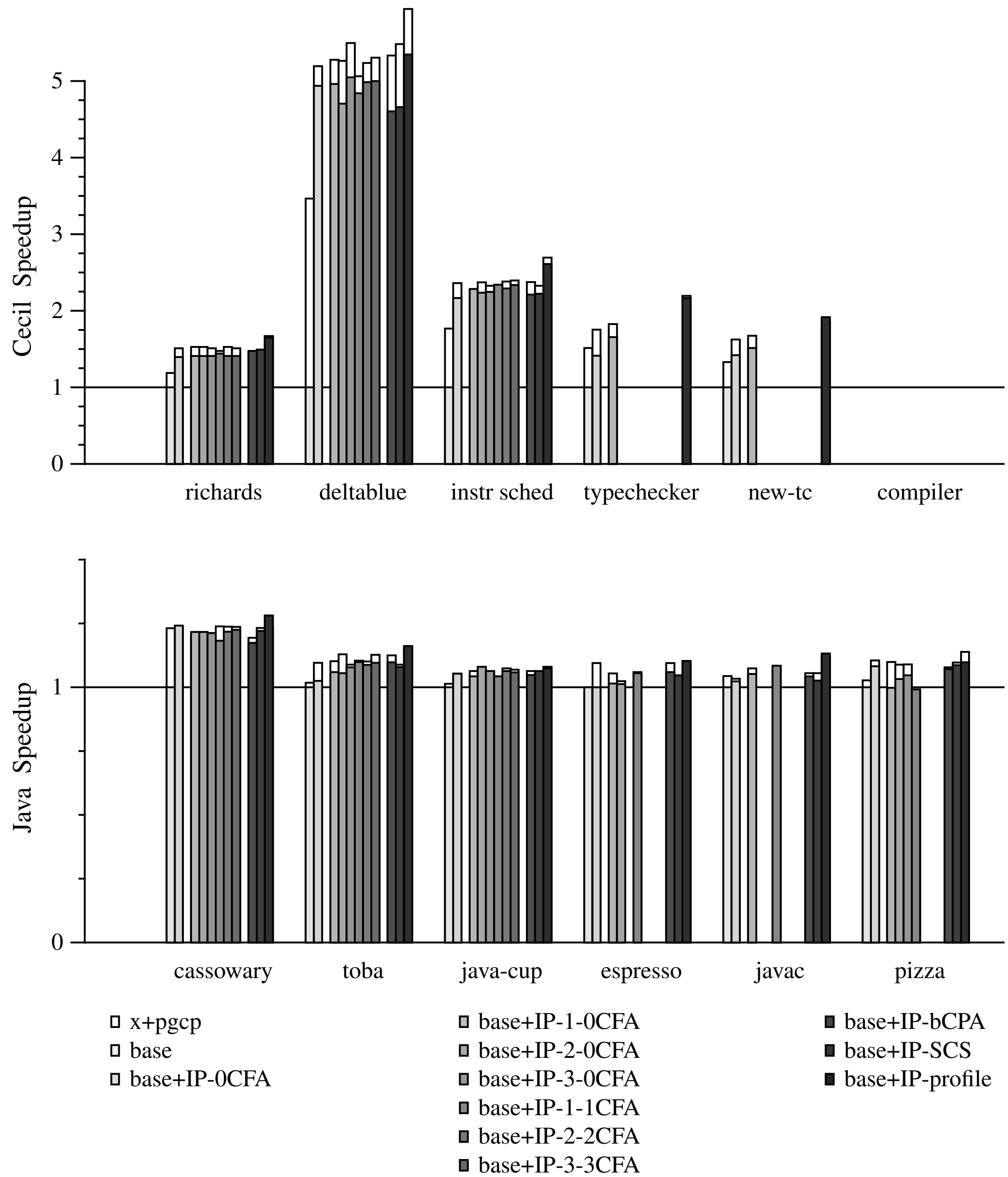


Figure 4.6: Performance impact of context-sensitive algorithms

that further improvements may be possible if a scalable and more precise context-sensitive algorithm could be developed. Overall, at least for these programs, current context-sensitive call graph construction algorithms are not an attractive option. Either no significant speedups over 0-CFA were enabled, or call graph construction costs were prohibitively high.

4.5 Approximations of 0-CFA

The goal of the context-sensitive algorithms discussed in the previous section is to build more precise call graphs than those built by 0-CFA, thus enabling more effective interprocedural analysis and larger bottom-line application performance improvements. Although the context-sensitive algorithms do enable significant performance improvements beyond those enabled by 0-CFA for a few of the benchmarks, they do so at a prohibitively high cost in call graph construction time. In fact, most of the studied context-sensitive algorithms cannot be practically applied to Cecil or Smalltalk programs that contain more than a few thousands lines of code. There was less of a scalability problem for the Java benchmarks, although the brute force approach of *k-l*-CFA was less successful. The algorithms presented in this section take the opposite approach: rather than trying to improve the precision of 0-CFA, they attempt to substantially reduce call graph construction time while preserving as much as possible of the bottom-line performance benefits obtained by 0-CFA.

The design and an initial implementation of these algorithms in the Vortex compiler was done by DeFouw [DeFouw et al. 98]. The experimental results in this dissertation are based on my new implementation of the algorithms that fully integrates them into Vortex's call graph construction implementation framework and that reduces call graph construction times by a factor of two to four and memory requirements by one to two decimal orders of magnitude.

4.5.1 Algorithm Descriptions

The 0-CFA algorithm (section 4.3) only generates inclusion constraints. The basic constraint satisfaction method for inclusion constraints is to simply propagate class contour information from sources to sinks in the program dataflow graph until a fix-point is reached. This solution method results in a worse case time complexity of $O(N^3)$, where N is a measure of program size. This follows from the fact that there will be $O(N^2)$ edges in the dataflow graph and $O(N)$ class contours may need to be propagated along each edge. Heintze and McAllester describe an alternative solution method for 0-CFA, based on constructing the subtransitive control flow graph, that requires only $O(N)$ time to compute a solution. However, it applies only to programs with bounded-size types [Heintze & McAllester 97]. In contrast to subtransitive 0-CFA, the approximation techniques defined in this section are generally applicable to any program, but yield less precise analyses than 0-CFA.

4.5.1.1 Unification

The first approximation is to replace the inclusion constraints generated by 0-CFA with bounded inclusion constraints, $\supseteq_p$, where p is some constant non-negative integer. The two families of algorithms defined in section 4.5.1.3 use a uniform value of p throughout their constraint graph, but a number of other strategies are possible (section 4.7). As described previously (section 3.4.4.4), once p distinct class contours have been propagated across a bounded inclusion constraint with a bound of p, the source and sink nodes connected by the constraint are unified, thus guaranteeing that the constraint will be satisfied while avoiding any further propagation work between the two nodes. However, unifying two nodes may result in the reverse propagation of class contours through the dataflow graph, degrading the precision of the final analysis result (unification makes the analysis partially data flow-insensitive).

Equality constraints, which unify nodes as soon as they are connected by dataflow, have been used to define near-linear time algorithms for binding time analysis [Henglein 91] and alias analysis [Steensgaard 96]. Ashley explored reducing propagation costs by utilizing bounded inclusion constraints in a control flow analysis of Scheme programs utilized to support inlining [Ashley 96, Ashley 97]. In addition to approximating 0-CFA, he also applies the same technique to develop an approximate 1-CFA analysis. Shapiro and Horwitz have developed a family of alias analysis algorithms that mix propagation and unification to yield more precise results than Steensgaard's algorithm while still substantially improving on the O(N^3) complexity of a purely propagation-based algorithm [Shapiro & Horwitz 97]. Their approach is significantly different from the one used by Ashley and discussed in this dissertation. Instead of using bounded inclusion constraints, they randomly assign each node in the dataflow graph to one of k categories. If two nodes in the same category are connected by an edge they are immediately unified, but nodes in different categories are never unified. Because the initial (random) assignment of nodes to categories can have a large effect on the final results of analysis, they propose a second algorithm that runs their basic algorithm several times, each time with a different random assignment of nodes to categories.

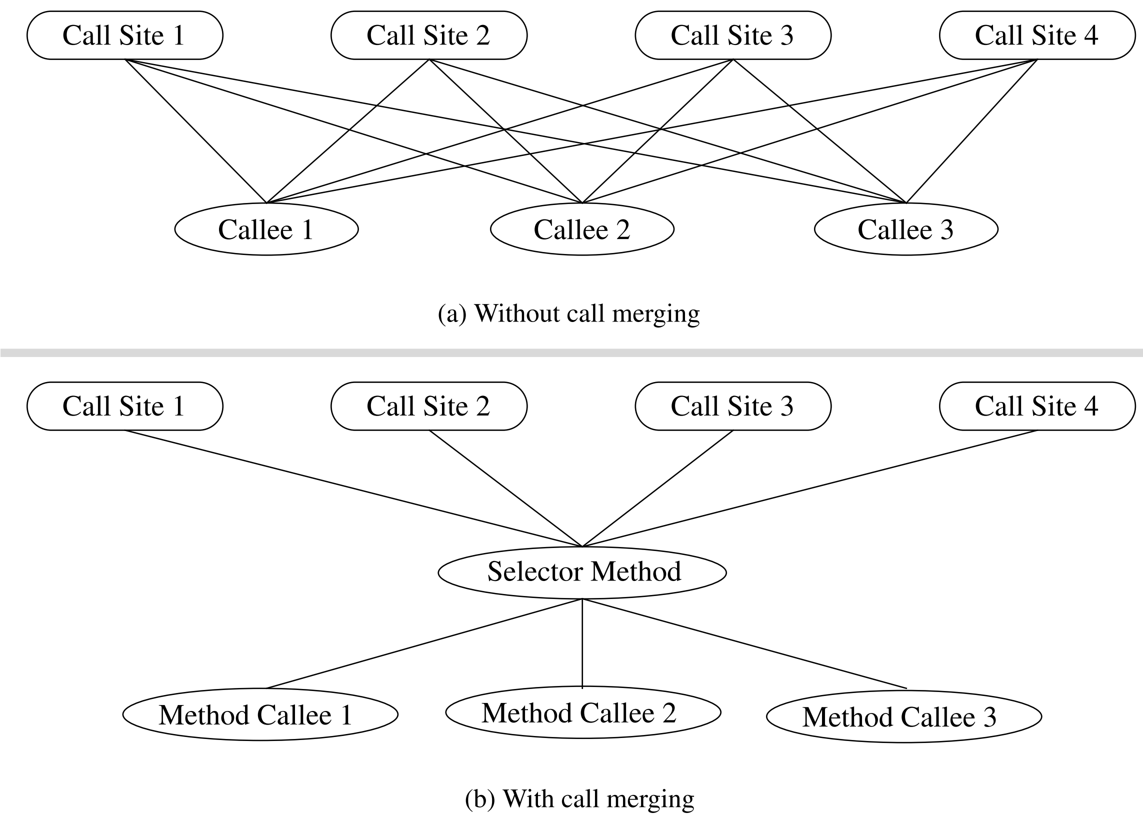


Figure 4.7: Dataflow graph without and with call merging

4.5.1.2 Call Merging

Call merging asymptotically reduces the number of edges in the dataflow graph by introducing a factoring node between the call sites and callee procedures of each selector.[7] Figure 4.7 illustrates call merging; figure 4.7(a) shows the unmerged connections required to connect four call sites with the same selector to three callee procedures with that selector and figure 4.7(b) shows the connections required if call merging is utilized. To simplify the picture, only a single edge is

7. A selector encodes the message name and number of arguments; for example, the selector for a send of the **`union`** message with two arguments is **`union/2`**. Note that because Vortex's front ends use name mangling to encode any static overloading of message names, selectors also encode some information from the source language's static type system.

shown connecting each node; in the actual dataflow graph this single edge would expand to a set of edges connecting each actual/formal pair and the return/result pair.

Call merging reduces the number of edges in the program dataflow graph from $O(N^2)$ to $O(N)$, and thus enables a reduction in worst case analysis time of $O(N)$ [DeFouw et al. 98]. However, this analysis time reduction may come at the cost of lost precision. In the merged dataflow graph, if a callee procedure is reachable from any one of a selector's call sites then it will be deemed reachable from all of the selector's call sites, leading to a potential dilution of the class contour sets and subsequent additional imprecisions in the final program call graph.

Call merging can be modeled as a pre-pass that transforms the program source prior to running the call graph construction algorithm. The pre-pass creates one method for each selector (with a new, unique name) that contains direct calls to all of the selector's methods. The result of the selector method is the union of the result of all of its "normal" callee methods. All call sites of the original program are replaced by direct calls to the appropriate selector method.

4.5.1.3 Algorithms

This dissertation examines two families of algorithms that approximate 0-CFA using the techniques described above. Both families of algorithms use a simplistic strategy of generating bounded inclusion constraints with a uniform constant value of p to control unification vs. propagation decisions. The two families of algorithms are distinguished by the presence or absence of call merging: the first family, p-Bounded, does not use call merging, whereas the second family, p-Bounded Linear-Edge, does.

For constant values of p, instances of the p-Bounded algorithm have a worst-case time complexity of $O(N^2\alpha(N,N))$[8] and instances of p-Bounded Linear-Edge have a worst-case time complexity of $O(N\alpha(N,N))$. If $p=\infty$, then no unification will be performed. The ∞–Bounded algorithm is exactly 0-CFA (no approximation will be performed) and has a worst case time complexity of $O(N^3)$. The ∞-Bounded Linear-Edge algorithm has a worst case time complexity of $O(N^2)$; DeFouw et al. called this degenerate case of p-Bounded Linear-Edge, Linear-Edge 0-CFA.

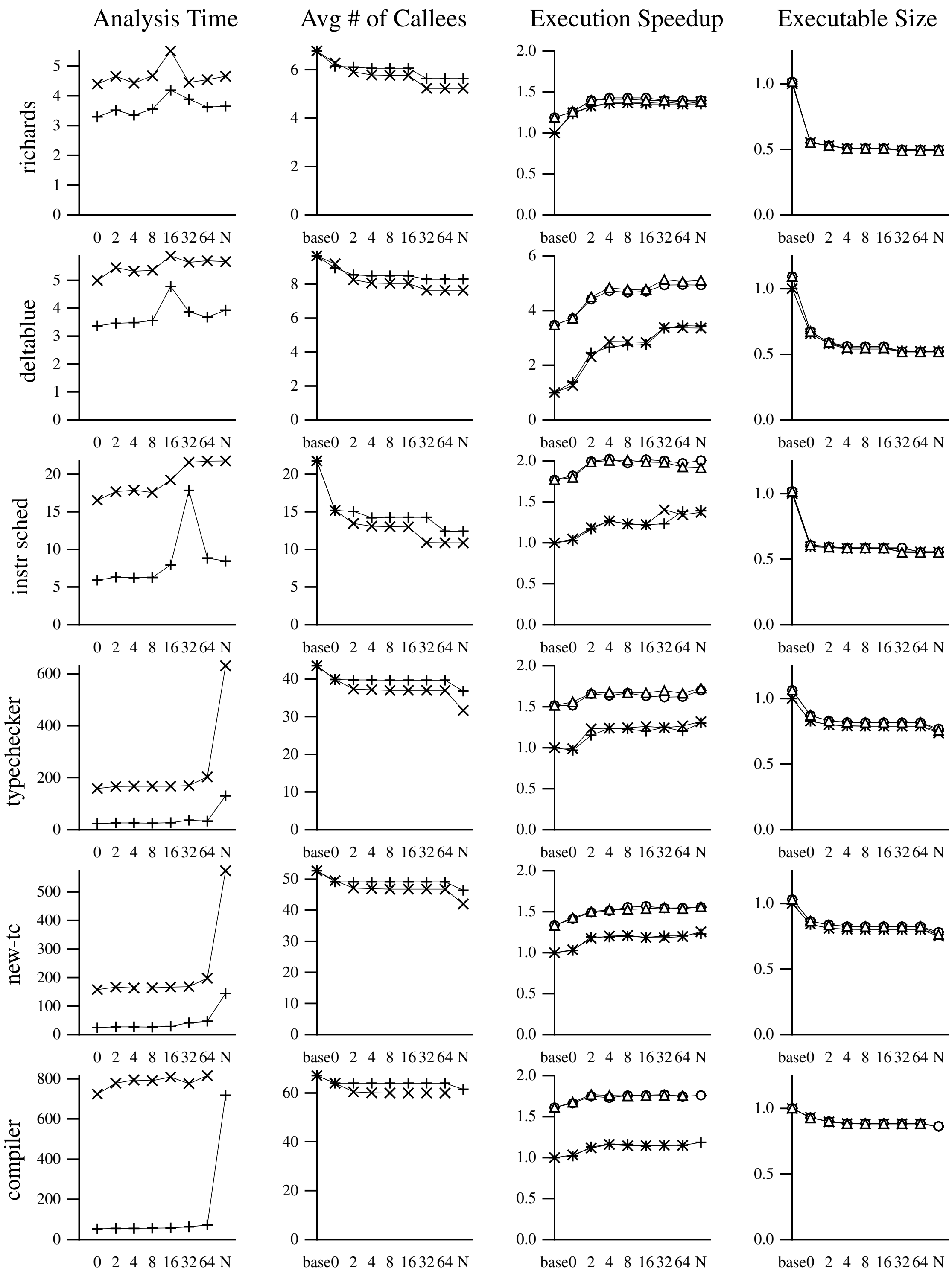
---

8. $\alpha(N,N)$ is the inverse Ackermann's function introduced by the fast union-find data structures. It is effectively a constant (for practical purposes less than 4)

The 0-Bounded Linear-Edge algorithm is very similar to Bacon and Sweeney's Rapid Type Analysis (RTA) [Bacon & Sweeney 96]. The key difference between the two algorithms is that RTA builds a single global set of live classes, whereas 0-Bounded Linear-Edge maintains a set of live classes for each disjoint region of the program's dataflow graph. Due to its simpler unification scheme, RTA is less precise than 0-Bounded Linear-Edge but has a slightly better worst-case time complexity of only O(*N*). However, with the exception of a few small programs, the theoretical differences between the two algorithms do not seem to result in any significant differences in either analysis time or bottom-line performance benefit [DeFouw et al. 98]. The versions of both families of algorithms assessed in this dissertation also include an additional approximation: rather than creating a unique class for each source level closure, all closures with the same number of arguments are considered to be instances of a single class, whose **`apply`** method is a union of the **`apply`** methods of the individual reachable closures with the appropriate number of arguments. Vortex implements both families of algorithms with and without this closure approximation, and experimental results (not included in this dissertation) have shown that this approximation of closures reduces analysis time by roughly a factor of two, while in almost all cases having no measurable impact on bottom-line application performance.

4.5.2 Experimental Assessment

The experimental results for the Cecil, Java, and Smalltalk benchmarks are shown by the series of four graphs per program found in figures 4.8, 4.9, and 4.10 respectively. Each graph plots two lines, one for *p*-Bounded and one for *p*-Bounded Linear-Edge, as *p* varies from 0 to ∞ (denoted as *N*) along the x-axis. Analysis cost is reported by the first graph in each series, which shows call graph construction time in CPU seconds. The second graph in each series focuses on the abstract precision of the resulting call graph by reporting the average number of callee procedures at a call site. The final two graphs in each series report the bottom-line performance benefits of interprocedural analysis using the constructed call graphs by reporting execution speed and executable size (normalized to **base**). In these final two graphs, lines are plotted for both the **base+IP** and **base+IP+pgcp** configurations. Memory usage is not shown. For constant values of *p*, memory was not a concern for the *p*-Bounded Linear Edge algorithm; for example the compiler program required between 25 and 35 MB. Memory was more of an issue for the *p*-Bounded



base+*p*-Bounded Linear Edge   base+*p*-Bounded Linear-Edge+ pgcp

base+*p*-Bounded   base+*p*-Bounded+pgcp

Figure 4.8: Approximations of 0-CFA (Cecil)

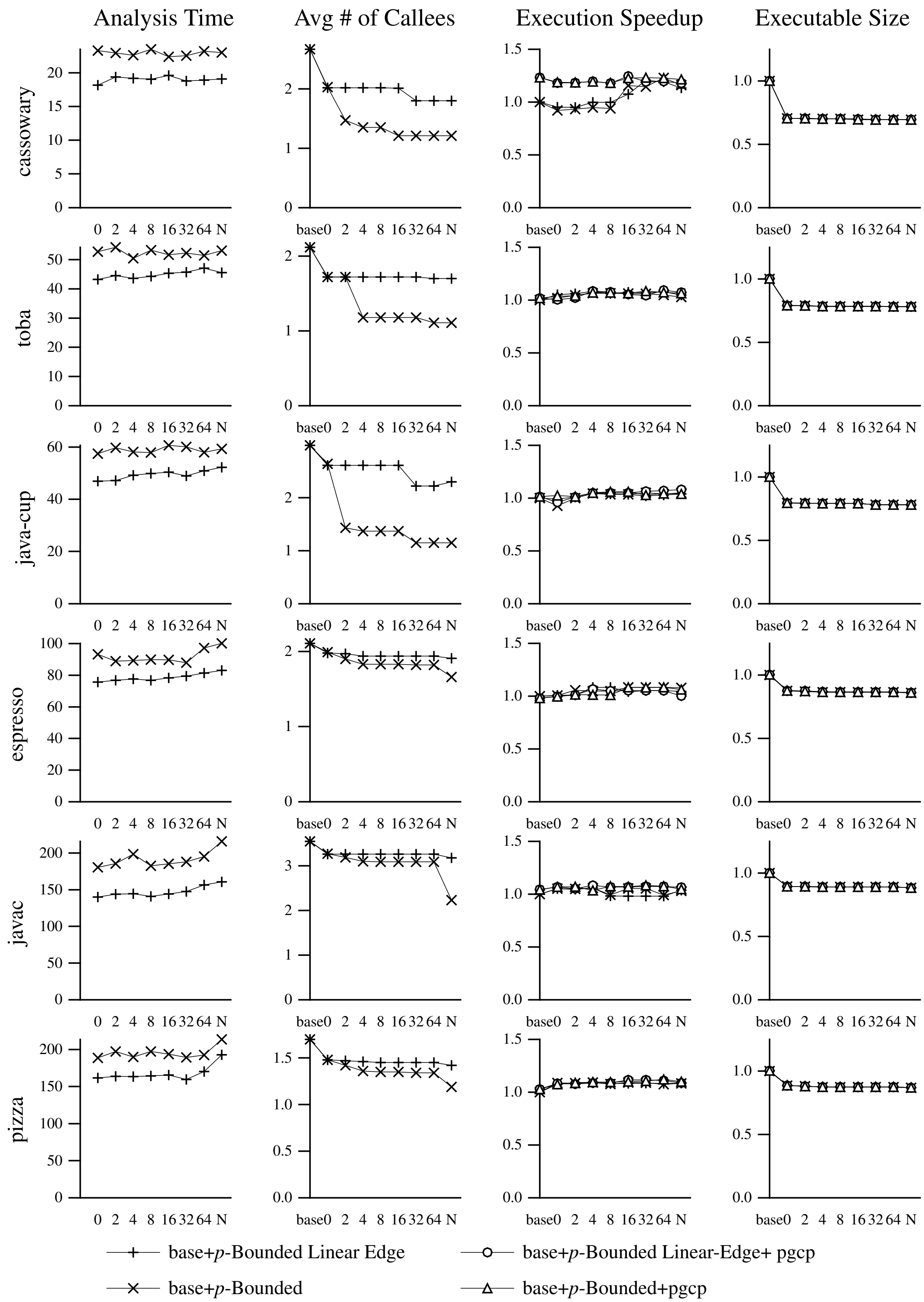


Figure 4.9: Approximations of 0-CFA (Java)

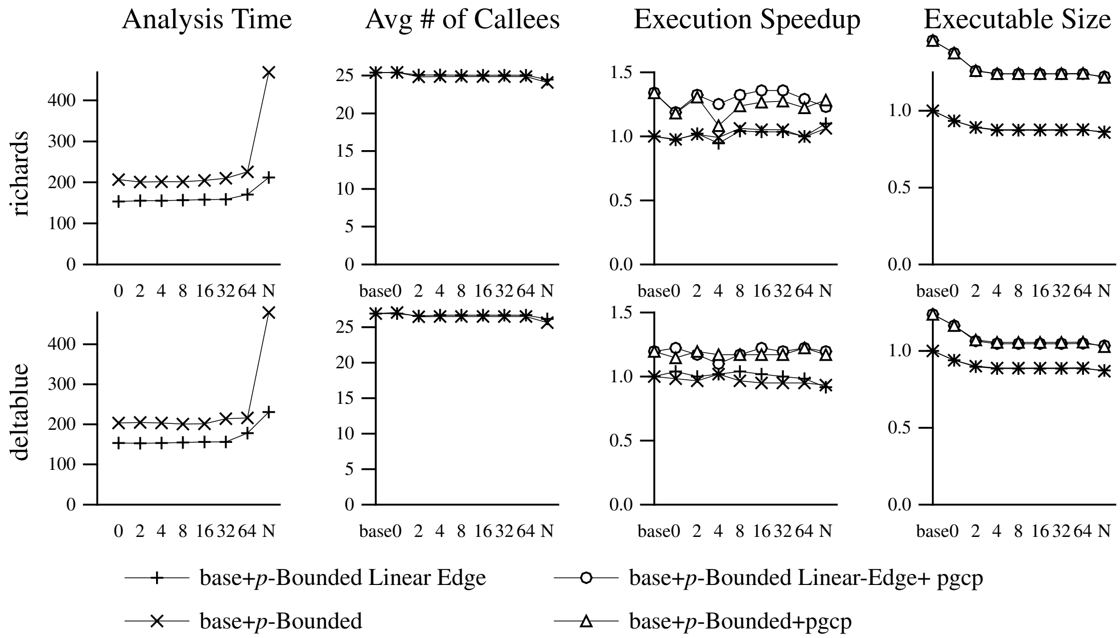


Figure 4.10: Approximations of 0-CFA (Smalltalk)

algorithm. For constant values of p, call graph construction for the compiler program consumed over 200 MB and for $p = \infty$ did not terminate (memory usage quickly grew to over 1GB, and after three days of thrashing the configuration was killed). For the typechecker and new-tc programs ∞-Bounded consumed just over 400 MB during call graph construction. Complete memory usage statistics can be found in appendix B.1.

Several major trends can be observed in the data. The call merging approximation does result in some degradation of call graph precision: for most of the programs, at the same value for p the p-Bounded call graph has a noticeably smaller average number of possible callee procedures per call site than the p-Bounded Linear Edge call graph. As expected, larger values of p do result in more precise call graphs, with the largest improvements in precision coming as p increases from 0 to a small constant (4 or 8). Surprisingly, across all the benchmarks the additional precision of the near-quadratic time p-Bounded algorithm does not enable any measurable performance improvements over that obtained by using the near-linear time p-Bounded Linear-Edge algorithm. However, as one would expect, the near-linear time algorithm is significantly faster and requires less memory, and therefore should be preferred. The additional precision enabled by using bounded inclusion constraints instead of equality constraints (p = a small constant vs. $p = 0$)

does translate into significant improvements in bottom-line application performance. The final major trend is the inverse-knee in the analysis times curves for intermediate values of p. This affect is present for almost all of the programs and is caused by the interactions between propagation and unification. As p increases, more propagation work is allowed along each edge in the dataflow graph, tending to increase analysis time. However, larger values of p also result in more precise call graphs, thus making the dataflow graph smaller by removing additional unreachable call graph nodes and edges and reducing analysis time. This second effect requires a moderate degree of propagation before it becomes significant (that gets larger as program size increases, and thus the knee gradually shifts to the right as programs increase in size). Thus, for intermediate values of p, the data flow graph does not become significantly smaller, but more propagation work can be incurred along edges between dataflow nodes that are eventually unified in any case.

4.6 Comparison of Algorithms

Table 4.4 contains a summary of the algorithms empirically assessed in this chapter. It contains the name of the algorithm, the section number in which it was presented, a canonical reference, and the values of the six parameters of the general algorithm that give rise to this algorithm instance.

Figure 4.11 depicts the relative precision of all the call graph construction algorithms described in this dissertation (some of which are *not* actually implemented in the Vortex compiler). Algorithm *A* is more precise than algorithm *B* if, for all input programs, G_A is at least as precise as G_B, and there exists some input program such that G_A is more precise than G_B. This (transitive) relationship is depicted by placing G_A above G_B and connecting them with a line segment.

All sound algorithms construct call graphs that are conservative approximations of G_{ideal}, the optimal sound call graph which is the greatest lower bound over all G_{prof_i}. All context-sensitive call graph construction algorithms produce call graphs at least as precise as those produced by 0-CFA, and thus are depicted above it in the lattice. As a convention, the k in various algorithm names stands for an arbitrarily large finite integer value; infinite values of a parameter are represented by ∞. The relationships between instances of the same parameterized family of algorithms is implied by the values of their parameters. For example, the k-l-CFA family of algorithms form an infinitely tall and infinitely wide sub-lattice. Increasing the degree of context-

Table 4.4: Algorithm summary

0-CFA	Section 4.3	[Shivers 88, Shivers 91]
$PKS(caller$: ProcContour, c : CallSite, a : AllTuples(ClassContourSet), $callee$: Procedure) $\rightarrow \{\perp\}$		
$IVKS(i$: InstVariable, b : ClassContourSet) $\rightarrow \{\perp\}$		
$CKS(c$: Class, p : ProcContour) $\rightarrow \{\perp\}$		
$EKS(c$: Closure, p : ProcContour) $\rightarrow \{\perp\}$		
$CIF : p = \infty, SIF : \varnothing$		
k-l-CFA	Section 4.4.1.1	[Shivers 88, Shivers 91]
$PKS(caller$: ProcContour, $c : CS$, a : AT(CCS), $callee$: P) $\rightarrow \{F_k(Proc(ID(caller)), Key(ID(caller)))\}$		
$IVKS(i$: InstVariable, b : ClassContourSet) $\rightarrow \begin{cases} \{\perp\}, l = 0 \\ b, \text{otherwise} \end{cases}$		
$CKS(c$: Class, p : ProcContour) $\rightarrow \begin{cases} \{\perp\}, l = 0 \\ \{\langle F_l(Proc(ID(p)), Key(ID(p)))\rangle\}, \text{otherwise} \end{cases}$		
$EKS(c$: Closure, p : ProcContour) $\rightarrow \begin{cases} \{\perp\}, \text{if } c \text{ contains no non-global free variables} \\ \{Key(ID(p)) \oplus lc \mid lc \in Lex(ID(p))\}, \text{otherwise} \end{cases}$		
$CIF : p = \infty, SIF : \varnothing$		
CPA	Section 4.4.1.2	[Agesen 95]
$PKS(caller$: ProcContour, $c : CS$, a : AllTuples(ClassContourSet), $callee$: Proc) $\rightarrow CPA(a)$ where $CPA(\langle S_1, S_2, \ldots, S_n\rangle) = S_1 \times S_2 \times \ldots \times S_n$		
$IVKS(i$: InstVariable, b : ClassContourSet) $\rightarrow \{\perp\}$		
$CKS(c$: Class, p : ProcContour) $\rightarrow \{\perp\}$		
$EKS(c$: Closure, p : ProcContour) $\rightarrow \begin{cases} \{\perp\}, \text{if } c \text{ contains no non-global free variables} \\ \{Key(ID(p)) \oplus lc \mid lc \in Lex(ID(p))\}, \text{otherwise} \end{cases}$		
$CIF : p = \infty, SIF : \varnothing$		
SCS	Section 4.4.1.2	[Grove et al. 97]
$PKS(caller$: ProcContour, $c : CS$, a : AllTuples(ClassContourSet), $callee$: Proc) $\rightarrow \{a\}$		
$IVKS(i$: InstVariable, b : ClassContourSet) $\rightarrow \{\perp\}$		
$CKS(c$: Class, p : ProcContour) $\rightarrow \{\perp\}$		
$EKS(c$: Closure, p : ProcContour) $\rightarrow \begin{cases} \{\perp\}, \text{if } c \text{ contains no non-global free variables} \\ \{Key(ID(p)) \oplus lc \mid lc \in Lex(ID(p))\}, \text{otherwise} \end{cases}$		
$CIF : p = \infty, SIF : \varnothing$		
p-Bounded	Section 4.5	[DeFouw et al. 98]
$PKS(caller$: ProcContour, c : CallSite, a : AllTuples(ClassContourSet), $callee$: Procedure) $\rightarrow \{\perp\}$		
$IVKS(i$: InstVariable, b : ClassContourSet) $\rightarrow \{\perp\}$		
$CKS(c$: Class, p : ProcContour) $\rightarrow \{\perp\}$		
$EKS(c : Closure, p : ProcContour) \rightarrow \{\perp\}$		
$CIF : p = p, SIF : \varnothing$		
p-Bounded Linear Edge	Section 4.5	[DeFouw et al. 98]
Same as p-Bounded, with the addition of the call merging approximation,		

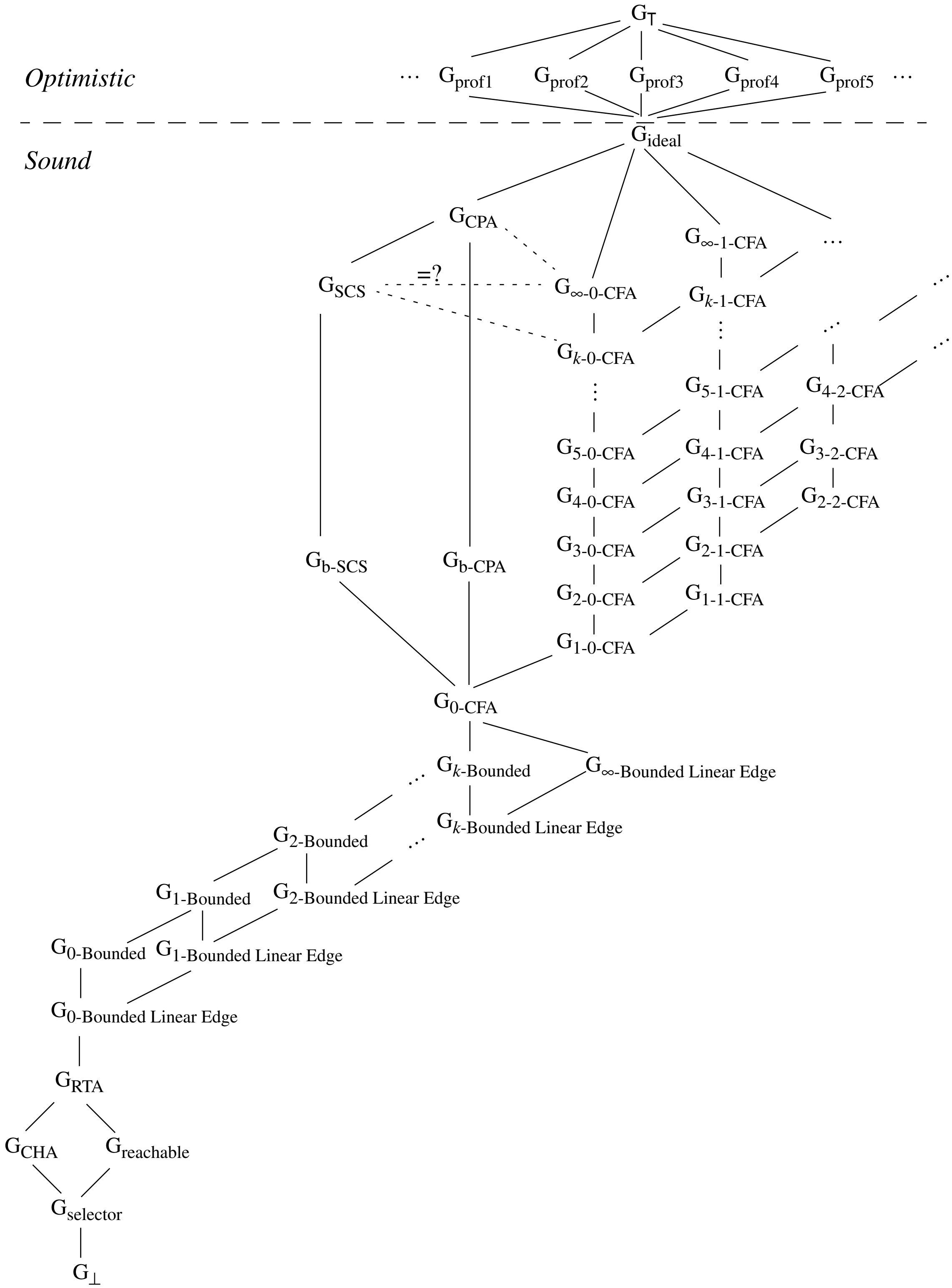


Figure 4.11: Relative precision of computed call graphs

sensitivity in the analysis of data structures and procedures improves call graph precision along independent axes, therefore there is no relationship between pairs of algorithms such as 2-0-CFA and 1-1-CFA. Similarly, the *p*-Bounded and *p*-Bounded Linear-Edge families of algorithms form two parallel infinitely tall sub-lattices positioned between RTA and 0-CFA. The implications of call merging and unification are different, and thus there is no relationship between *n*-Bounded and (*n+1*)-Bounded Linear-Edge; there exist programs for which each produces a more precise call graph than the other.

One of the most interesting portions of the lattice is that depicting the relationships among CPA, SCS, *k*-0-CFA, and ∞-0-CFA. Unfortunately, the lattice-theoretic definition of call graphs presented in section 3.2 cannot be used to determine the relationship between pairs of algorithms with incomparable *ProcKey* partial orders, so this discussion is informal and somewhat speculative (indicated by dotted lines in figure 4.11). All four of these algorithms use the same (context-insensitive) strategy to analyze data structures. Therefore, any precision differences will be due to their handling of polymorphic functions. By itself, context-sensitive analysis of functions can only increase call graph precision by increasing the precision of some procedure's formal class sets by creating distinct contours to distinguish between two call sites with different argument class sets. By definition, CPA and SCS create new contours whenever argument class sets differ, so they should construct call graphs that are at least as precise as those generated by either *k*-0-CFA or ∞-0-CFA. More interestingly, there are programs for which SCS and/or CPA generate call graphs that are more precise than those produced by *k*-0-CFA or even ∞-0-CFA. Two such programs are shown in figure 4.12. Figure 4.12(b) depicts a program in which SCS, CPA, and ∞-0-CFA compute more precise call graphs than *k*-0-CFA. The *k+1* levels of wrapper procedures result in *k*-0-CFA determining that the **`+(@num,num)`** method could be invoked from the call site in **`wrap_k`**. None of the other three algorithms make this mistake. Figure 4.12(c) repeats the example from section 4.4.1.2 that was used to illustrate how CPA can be more precise then SCS. Applying SCS, *k*-0-CFA or ∞-0-CFA to this program results in a call graph in which **`+(@num,num)`** is a possible callee of **`double`**. The CPA call graph does not contain this inaccuracy. Therefore, I believe that it is likely that CPA is more precise than the other three algorithms, and that SCS and ∞-0-CFA are equally precise.

```
class num;
   method +(@num,@num){
      ....
   }
   method double(x@num) {
      return x + x;
   }

class float inherits num;
   method +(@float,@float){
      ....
   }

class int inherits num;
   method +(@int,@int){
      ....
   }
```

(a) Class hierarchy

```
main() {
   test1()
   test2()
}
test1() {
   wrap_0(5, 5);
}
test2() {
   wrap_0(3.5, 3.5);
}
wrap_0(x,y) {
   return wrap_1(x,y);
}
wrap_1(x,y) {
   return wrap_2(x,y);
}
....
wrap_k(x,y) {
   return x + y;
}
```

(b) Program one

```
main() {
   num x;
   x := 3;
   if (random()) {
      x := 3.5;
   }
   return double(x);
}
```

(c) Program two

Figure 4.12: Example programs for call chain vs. parameter context-sensitivity

4.7 Future Work

There are two main avenues for future work in the general areas discussed in this chapter. First, there are a number of potentially interesting unexplored (or under-explored) regions in the design space of call graph construction algorithms for object-oriented languages. Second, additional client interprocedural analyses could be designed and/or implemented to further exploit the program call graph resulting in an increased bottom-line benefit.

For the majority of the benchmarks, there was a significant gap between the performance obtained by even the most successful call graph construction algorithms and the upper bound represented by G_{prof}. In theory, new context-sensitive algorithms might narrow this performance gap, but designing a more precise context-sensitive algorithm that is also scalable enough to be applied beyond the realm of toy benchmark programs may prove to be an insurmountable challenge. A more promising region of the algorithm design space is the combination of context-sensitivity with unification-based approximations to bound analysis time; such algorithms may be able to exceed the performance benefits of 0-CFA while still consuming fewer analysis time resources. Also, more exact unification-based approximations of 0-CFA could be developed by designing more sophisticated unification control strategies. For example, profile data could be used to identify performance-critical regions of the program that could then be given higher unification thresholds or some form of global budgeting of propagation (as opposed to the local, per-edge budgeting described in section 4.5.1.3) could be developed. Finally, the idea of *reflow analysis* [Shivers 91] or *non-monotonic improvement* [Grove et al. 97] could be exploited to quickly build an initial imprecise call graph and then iteratively improve it. This idea is the basis of Plevyak's iterative algorithm [Plevyak & Chien 94], which has been successfully applied to small (2000 lines of code) Concert programs. The initial iteration of the iterative algorithm is roughly equivalent to CPA, which our experiments have shown does not scale to large Cecil programs. Therefore it is unlikely that an unmodified version of Plevyak's algorithm will be applicable to large Cecil programs. However, it might be possible to develop more scalable iterative algorithms.

A second avenue of further exploration is the design and/or implementation of additional interprocedural analyses to increase the performance benefits of constructing the call graph. The performance results presented in this chapter are based on using five particular interprocedural analyses; additional interprocedural analyses could magnify bottom-line performance improvements. For example, object-inlining [Dolby 97] could be applied to reduce the memory footprint and memory access costs of object-oriented programs, and an interprocedural analysis to detect redundant or unnecessary synchronization operations might substantially improve the performance of Java programs.[9]

4.8 Other Related Work

Much of the related work in call graph construction algorithms for object-oriented and functional languages was already discussed in this chapter. Therefore, this section focuses on call graph

9. The version of Vortex used in this dissertation does not support multithreading. However, thread support has been recently added to Vortex's runtime system, and current work by other members of the Cecil/Vortex group includes exploring the possibility of using interprocedural analysis to improve the performance of multithreaded Java programs.

```
(let ([f (lambda (x) (lambda (y) x))])
   ((f 0) (f #t)))


((lambda (f) ((f 0) (f #t)))
   (lambda (x) (lambda (y) x)))
```

Figure 4.13: Polymorphic splitting example

construction algorithms that are not easily expressible in the general algorithm framework of chapter 3 and thus did not naturally fit into a previous section of this chapter.

One such algorithm is polymorphic splitting [Wright & Jagannathan 98]. Polymorphic splitting is a control flow analysis that relies on the syntactic clues provided by let-expressions to guide its context-sensitivity decisions; since let-expressions are not included in the core object-oriented language defined in section 3.3.1, the analysis of section 3.3.4 does not define how to analyze them. However, our analysis framework could easily be extended to support polymorphic splitting by incorporating the rules of Wright and Jagannathan. Intuitively, polymorphic splitting works by ensuring that each reference to a let-bound function value is analyzed in a different contour. Figure 4.13 contains an example from section 3.5 of [Wright & Jagannathan 98] that illustrates the strengths and weaknesses of polymorphic splitting. The two expressions are semantically equivalent, but although polymorphic splitting is able to determine that the first expression evaluates to **`{number}`**, it can only determine that the second expression evaluates to **`{number, true}`**, due to the lack of syntactic clues to guide its context-sensitivity decisions. As this (contrived) example illustrates, polymorphic splitting will be successful in analyzing programs in which polymorphic code is "tagged" by explicit let statements, but will fail on programs that do not follow this convention.

The analysis framework of chapter 3 also does not completely encompass algorithms that do not monotonically converge on their final solution. This class of algorithms are often referred to as iterative algorithms, since they can be viewed as iterating between phases of monotonic and non-monotonic behavior. In previous work, we informally defined an algorithmic framework that encompassed both iterative and non-iterative call graph construction algorithms [Grove et al. 97]. However, only non-iterative algorithms are currently supported in the Vortex implementation framework. In the terminology of this previous work, the analysis framework of this dissertation is simply a precise definition of the Monotonic Refinement component of a general call graph construction algorithm. Iterative algorithms add a Non-Monotonic Improvement component that allows them to discard pieces of an intermediate solution call graph and re-do portions of the analysis to compute a more precise final call graph. Iterative algorithms are derived from Shivers's proposal for reflow analysis [Shivers 91]. Currently, the only implemented iterative algorithm that I know of is Plevyak's iterative algorithm [Plevyak & Chien 94, Plevyak 96]. In previous work, we proposed one possible approach for designing a less aggressive iterative algorithm based on *exact unions* [Grove et al. 97]. In a sound call graph, each variable's class contour set must be a superset of all the class contour sets associated with inflowing data flow edges. However, algorithms that work by refining a pessimistic initial call graph, such as $G_{selector}$, often lead to class contour sets that are proper supersets of the union of its inflowing class sets. Such a call graph can be improved without making it unsound by replacing each variable's class contour set with the exact union of its inflowing class contour sets. This narrowing may enable further downstream narrowings and removals of unreachable call arcs. This process can continue until a fix-point is reached, or, since the call graph is sound, can be terminated if an analysis-time budget is exceeded.

A number of the papers proposing new call graph construction algorithms have empirically assessed the effectiveness of their algorithm(s) by implementing them in an optimizing compiler and using the resulting call graphs to perform one or more interprocedural analyses. In some cases comparisons are made against other call graph construction algorithms, while other papers simply compare against a baseline system which performs no interprocedural analysis. One empirical assessment of interprocedural analysis that is somewhat different than the standard set of experiments is Hölzle and Agesen's comparison of the effectiveness of interprocedural class analysis based on the Cartesian Product Algorithm and profile-guided class prediction for the optimization of Self programs [Hölzle & Agesen 96]. They found that there was very little performance difference (less than 15%) between three optimizing configurations: one using only profile-guided class prediction, one using only interprocedural class analysis, and one using both techniques. Our results in Cecil, a language approximately equivalent to Self in terms of the challenges it presents to an optimizing compiler, were somewhat different. We found that for small programs interprocedural class analysis dominated profile-guided class prediction. On the larger programs, in isolation profile-guided class prediction enabled larger speedups than interprocedural class analysis, but the combination of the two enabled larger speedups than either technique alone.

4.9 Summary

To summarize the costs and benefits of using different call graph construction algorithms as the basis for interprocedural analysis and subsequent optimization, experimental results for a subset of the algorithms assessed in this chapter are repeated in figures 4.14, 4.15, and 4.16. The $G_{selector}$ and G_{prof} algorithms (section 4.2) represent lower and upper bounds on the performance impact achievable through interprocedural analysis. The 8-Bounded Linear-Edge (section 4.5), 0-CFA (section 4.3), and SCS (section 4.4) algorithms each represent substantially different regions in the algorithmic design space ranging from fairly fast but imprecise algorithms to slow but fairly precise algorithms. The choice of $p = 8$ for the p-Bounded Linear Edge algorithm was made based on the performance of the algorithm on the largest Cecil benchmarks. Optimal values for p vary from program to program (for example, on cassowary $p = 8$ results in virtually no improvement over base, but $p = 16$ shows some benefits and $p = 32$ almost matches 0-CFA), but for these benchmarks are typically $2 < p < 16$.

Figure 4.14 reports the normalized execution speeds obtained by the **base+IP** and **base+IP+pgcp** configurations of these five call graph construction algorithms. The **base** and **base+pgcp** configurations are also shown for comparison. Figure 4.15 shows call graph construction costs plotted as a function of program size. In these programs, context-sensitive analysis did not enable significant performance improvements over 0-CFA, although call graph construction costs were often significantly higher in the context-sensitive algorithms. The 8-Bounded Linear-Edge algorithm only enabled a portion of the speedup of 0-CFA, but did so at a fraction of the analysis-time costs.

Figure 4.16 combines the previous two graphs by plotting execution speedup of the **base+IP** configuration as a function of call graph construction time. To generate this graph, the 0-CFA algorithm was arbitrarily chosen as the unit of comparison. Then, for each benchmark program and algorithm pair, a point is plotted to show the cost/benefit of the algorithm relative to the cost/benefit of 0-CFA for that program. Only points whose x and y values are between 0 and 200 are

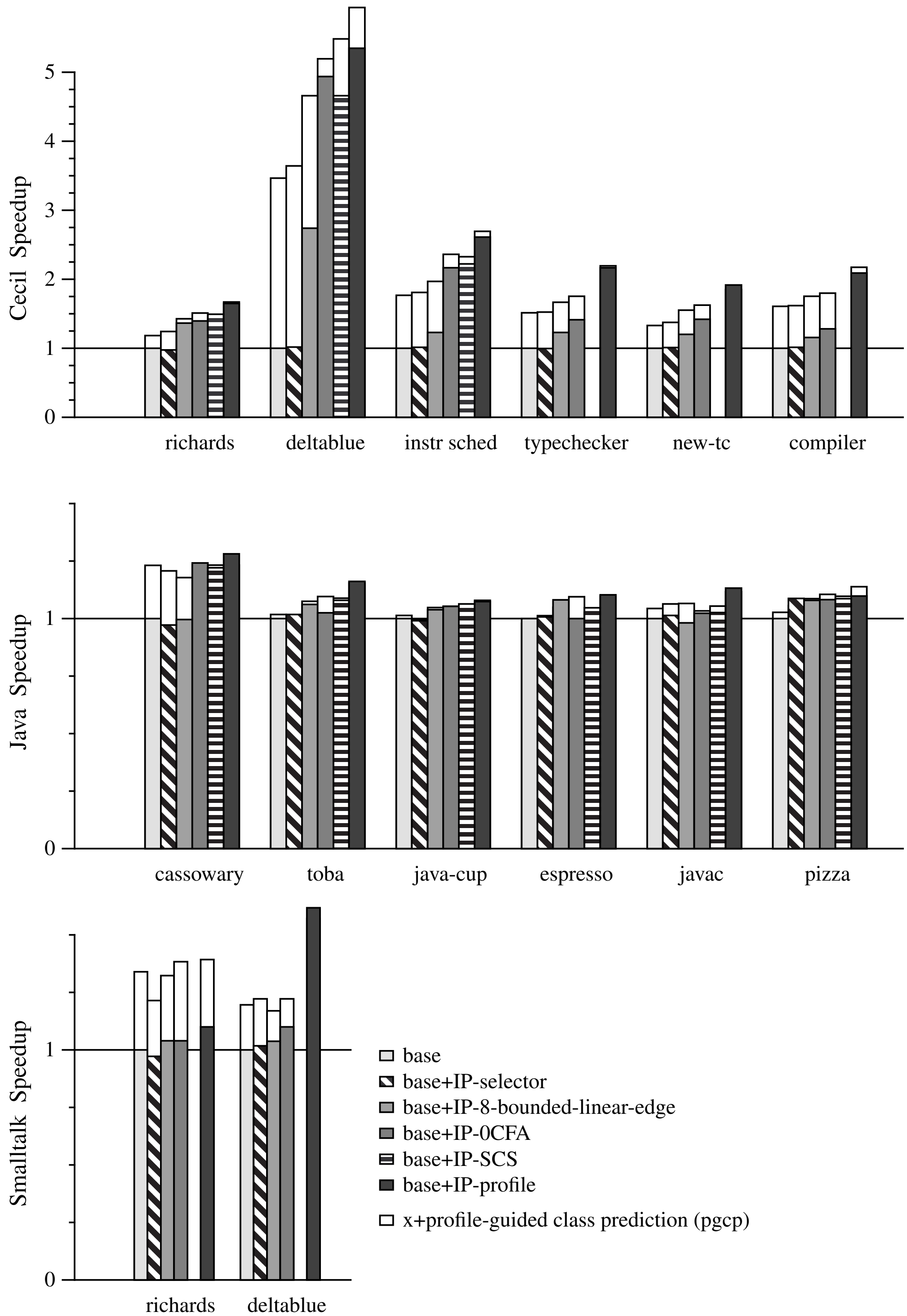


Figure 4.14: Execution speed summary

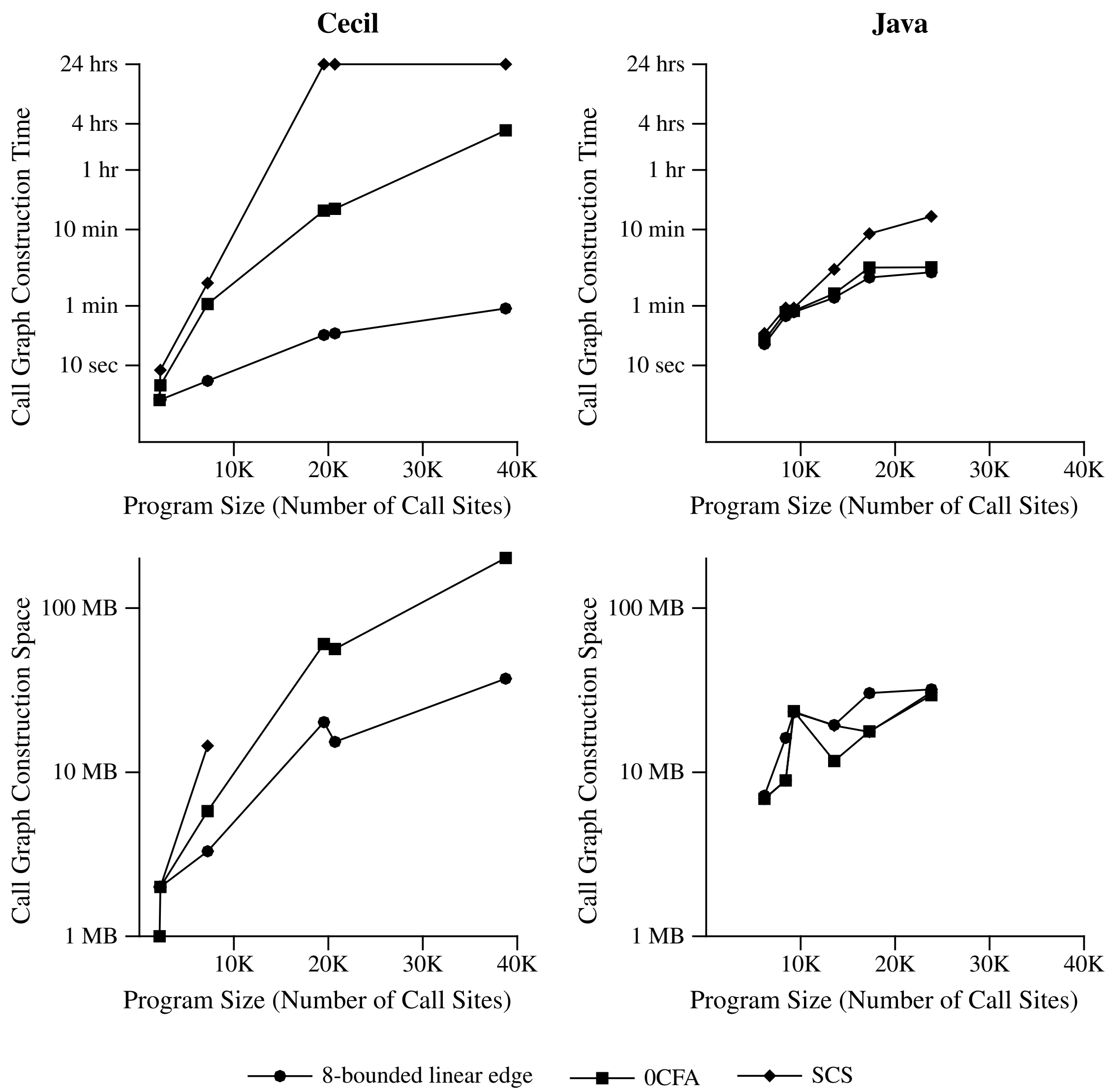


Figure 4.15: Call graph construction costs summary

---

shown. For example, on compiler 8-Bounded Linear Edge call graph construction took 55 seconds and enabled a 16% speedup over **base**; 0-CFA call graph construction took 11,781 seconds (3.25 hours) and enabled a 28% speedup over **base**. Therefore, a point is plotted for 8-Bounded Linear Edge at (0.5,57). The Cecil and Smalltalk plots are quite telling. For those programs on which it actually completed, the context-sensitive analysis (SCS) usually increased call graph construction costs with little benefit. For the Smalltalk programs and the three largest Cecil programs, the points corresponding to 8-Bounded Linear-Edge are clustered part way up the y-axis very close to

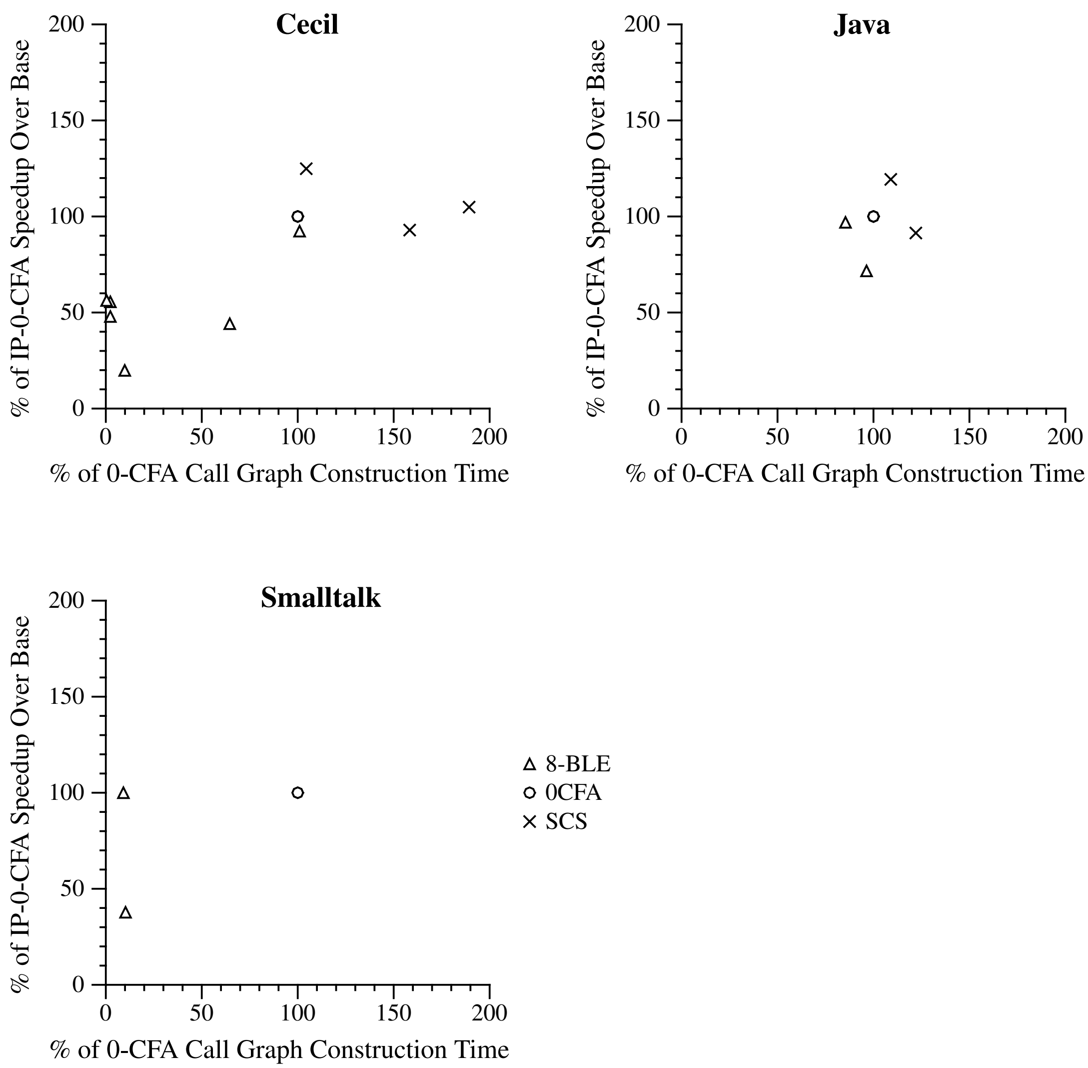


Figure 4.16: Cost/Benefit trade-offs of call graph construction algorithms

the axis: for these programs the near-linear time algorithm achieved a significant portion of the 0-CFA benefits at a very small fraction of the 0-CFA cost. For the Java programs, the results are less dramatic, but some of the same trends can be observed. Of the three algorithms, 8-Bounded Linear Edge was the fastest, and SCS was the slowest. SCS speedups were fairly equivalent to those enabled by 0-CFA. The performance results for 8-Bounded Linear Edge were mixed, but on all the Java programs some *p* value between 4 and 32 was sufficient to enable virtually all of the 0-CFA speedups.

At least for the purposes of interprocedural analysis in Vortex, the *p*-Bounded Linear Edge algorithm with p = a small constant clearly represents an excellent trade-off of analysis time and call graph precision. The algorithm is fast and scalable, and it produces call graphs that are sufficiently precise to enable significant speedups over an already highly optimized baseline configuration.

Chapter 5

Programming Environment Support for Interprocedural Analysis

The primary motivation for performing interprocedural analysis is to enable an optimizing compiler to more exactly model the behavior of non-inlined procedure calls, thus enabling it to more effectively optimize the program. As demonstrated by the experimental results of the previous chapter, exploiting interprocedural information can yield substantial improvements in bottom-line application performance. However, performing optimizations based on interprocedural analysis can complicate recompilation after source code changes. In a programming environment that supports true separate compilation, compilers are restricted to making safe but pessimistic assumptions about all parts of the program that are external to a source unit. This local view of the source unit makes recompilation after source changes simple: only those compiled units whose source units were modified need to be recompiled. In contrast, in the presence of interprocedural analysis the correctness of an optimization applied to code in one compiled unit may depend on the properties of a number of source units. If any of these source units change, the optimization may no longer be correct, and the dependent compiled unit must be recompiled. Thus, an important job of a programming environment for a compiler that incorporates interprocedural analysis is to track the relationships among source units, interprocedural summaries, and compiled units to ensure that the appropriate summaries and compiled units are invalidated after a source unit changes.

This chapter will discuss the issues in fairly generic terms such as source unit and compiled unit. The granularity of these units is a critical design choice. For example, a source unit could be an entire source file, a class and all of its methods, or just single source declaration. Similarly, a compiled unit could contain the compiled code for all the declarations in a source file, for a single class and its methods, or just a single source declaration. Designing a dependency system involves making a number of granularity-related design choices that together determine the *dependency maintenance costs*, *invalidation processing costs*, and *selectivity* of the overall system. Dependency maintenance costs consist of the time required to construct and update any data structures that encode inter-unit dependencies and the space consumed by these data structures. Invalidation processing is any work that must be performed after the set of modified source units is identified to determine which of the compiled units is out-of-date (invalid) and must therefore be recompiled. Finally, a design's selectivity is determined by the expected number of compiled units that are out-of-date after a source unit changes. Selectivity is inversely related to recompilation costs: more selective designs invalidate fewer compiled units and thus incur lower recompilation costs. As a general rule, finer-grained designs tend to increase both selectivity (good), and dependency maintenance costs (bad). The relationship between granularity and the cost of invalidation processing is less clear. For a given program, finer-grained designs will have more dependency units, and thus tend to have a larger total number of inter-unit dependencies that may need to be processed. However, because each dependency unit is representing a smaller entity, the number of other units that depend on it (or that it depends on) may be reduced relative to a coarser-grained design. On the other hand, the expected number of dependent units per unit may increase in a finer-grained design because several formerly grouped entities have been split into multiple units, each with similar (or identical) dependencies. To be practical, a design for a dependency system must embody granularity decisions that balance maintenance, invalidation, and recompilation costs.

One straightforward approach for enabling interprocedural analysis to coexist with program evolution is to simply reanalyze and recompile the entire program after every source change. This coarse dependency strategy (every compiled unit depends on every source unit) is simple to implement and understand, but because of its high recompilation costs is only practical in fairly restricted domains. For example, the coarse strategy is likely to be acceptable when creating a final version of an application after development has finished, but is unlikely to be viable as part of the more interactive edit-compile-debug loop of normal day-to-day program development. Incorporating interprocedural analysis in a programming environment that is intended to support rapid turnaround after program changes is a much more challenging task: either interprocedural analysis, optimization, and compilation all need to be fast enough that they can be done from scratch after every program change (an unlikely prospect for sizeable programs), or the programming environment needs to support finer-grained incremental reanalysis and recompilation.

This chapter presents a design for supporting incremental interprocedural analysis and recompilation in the Vortex programming environment. A novel aspect of the design is its use of an implicit representation for several major categories of dependencies, thus substantially reducing expected dependency maintenance costs without impacting selectivity or invalidation processing costs. The overall design is compatible with any instantiation of the Vortex implementation framework described in section 3.4 (including all those empirically assessed in chapter 4), although some details of section 5.1.2 are dependent on whether an algorithm uses an implicit or explicit representation of the program's dataflow graph (see section 3.4.3.1). The next section overviews the general architecture of the proposed dependency system, reviews the existing Vortex dependency mechanisms, discusses some of the issues in supporting incremental call graph construction and interprocedural analysis in Vortex, and summarizes the current status of the implementation. The chapter concludes by discussing related work.

5.1 A Framework for Incremental Reanalysis

Figure 5.1 depicts the logical structure of the dependencies introduced by performing interprocedural analysis and optimization in the Vortex compiler infrastructure. Each node in the graph represents a specific kind of abstract information, such as a source unit, an interprocedural summary, or an aspect of the call graph such as its nodes or edges. A directed edge from **`A`** to **`B`** indicates that **`B`** consumes, and thus is dependent on, the information represented by **`A`**. Note that this diagram only shows the subset of Vortex's dependency structure that is directly due to call graph construction and call graph-based interprocedural analysis. In previous work, we described and empirically evaluated the dependency system used by Vortex to support class hierarchy analysis, compile-time method lookup, and cross-source unit inlining [Chambers et al. 95].

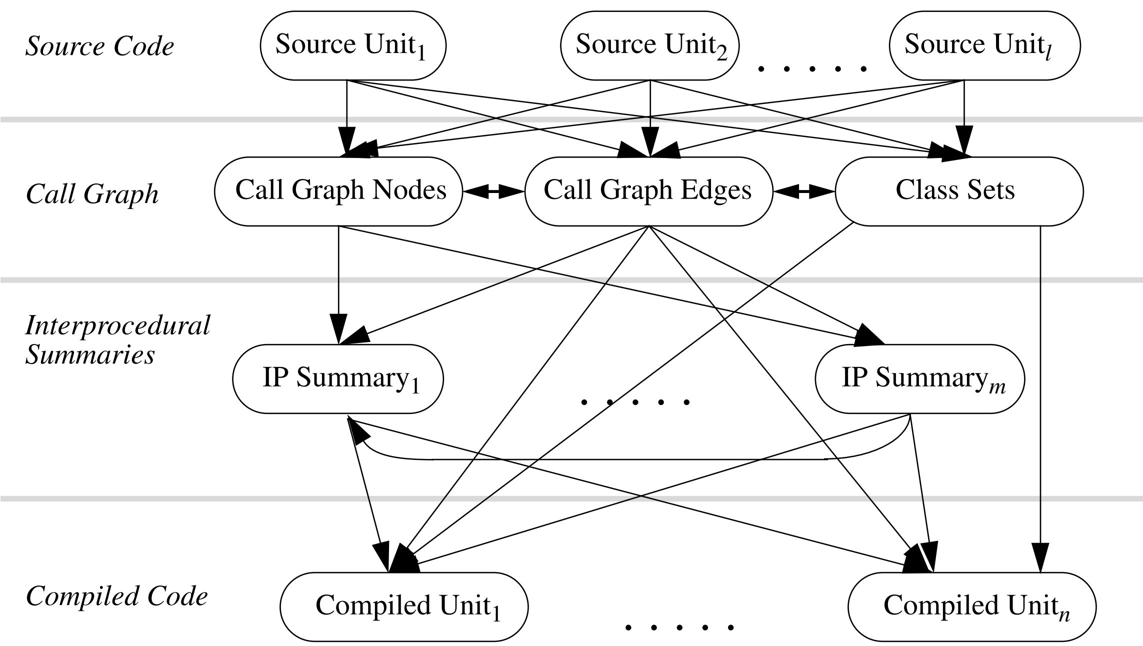


Figure 5.1: Dependency structure induced by interprocedural analysis

The information represented by the nodes in this dependency structure can be divided into four categories:

- *Source code*: Nodes in the source code level represent input source units. Exactly how much of the source program makes up a single source unit will affect both the dependency's selectivity and representation cost. Vortex treats each source declaration as a separate source unit. Other coarser-grained choices are possible; for example, a single source unit could include all the declarations in a class, module, or file.

- *Call graph*: The program call graph is the abstraction of the source program that Vortex presents to its interprocedural analyses. The program call graph consists of three mutually dependent components: call graph nodes, call graph edges, and class sets. The general algorithm of chapter 3 constructs a contour call graph that encodes a particular set of context-sensitivity decisions made by the specific interprocedural class analysis algorithm used to construct the call graph. Because the context-sensitivity strategy used during call graph construction may not be appropriate for client interprocedural analyses, Vortex's interprocedural analysis framework [Chambers et al. 96a] automatically presents clients with a summarized procedure-level call graph and allows them to use their own independent context-sensitivity strategies. It is this procedure-level call graph that is utilized (and thus depended on) by Vortex's interprocedural analyses.

- *Interprocedural Summaries*: The end result of performing an analysis over the program call graph is an interprocedural summary. The exact composition of a summary is analysis-specific (for example, a summary for interprocedural constant propagation might consist of a mapping from program variables to constants while a summary for interprocedural alias analysis might consist of a set of alias pairs), but all summaries serve the same function in the dependency structure: they encapsulate the results of an analysis and export only the "interesting" results to the rest of the system. One interprocedural summary may depend on another summary; for example, in Vortex, flow-sensitive interprocedural analyses such as constant propagation can use the results of MOD analysis to improve their precision. Portions of the call graph can also serve as interprocedural summaries; for example, Vortex uses the class set and call graph edge information to improve the results of intraprocedural class analysis and inlining.

- *Compiled Code*: Nodes at this level represent the final output of the compiler: compiled units of code. Vortex generates code at a fairly coarse granularity; each of its compiled units contains all of the code compiled for the declarations contained in an input source file.

Inter-level dependencies between nodes in each of the four categories are only introduced in the forward direction, and thus form a directed acyclic graph. Section 5.1.1 reviews the mechanisms used to represent and manipulate dependencies in Vortex. The subsequent three sections treat the dependencies that link each pair of adjacent levels. Each of these section examines one set of inter-level dependencies by answering three key questions:

- Where and why do dependencies exist between nodes in the two levels?

- How should the inter-level dependencies be represented?

- What actions should be taken to update the information represented by a client node when the information represented by a node on which it depends changes?

5.1.1 Existing Vortex Dependency Mechanisms

The Vortex compiler already includes a fairly sophisticated set of mechanisms for representing complex cross-source unit dependencies that were developed to support incremental recompilation in the presence of class hierarchy analysis and cross-source unit inlining. This dependency framework has been in daily use for over four years (Vortex is used as the compiler in its own development and both class hierarchy analysis and cross-source unit inlining are performed as a matter of course during normal development). A more detailed description and an empirical evaluation of Vortex's basic dependency mechanisms can be found elsewhere [Chambers et al. 95].

In Vortex's dependency framework, dependencies are represented by a directed, acyclic graph structure. Nodes in this graph represent information, including source units, compiled units, and interprocedural summaries, and an edge from one node to another indicates that the information represented by the target node is derived from or depends on the information represented by the source node. Depending on the number of incoming and outgoing edges, nodes are classified into three categories:

- *Source nodes* have only outgoing dependency edges. They represent information present in the source units comprising the program, such as the bodies of procedures and the class inheritance hierarchy.
- *Target nodes* have only incoming dependency edges. They represent information that is an end product of compilation, such as compiled units.
- *Internal nodes* have both incoming and outgoing edges. They represent information that is derived from some earlier information and used in the derivation of some later information.

The dependency graph is constructed incrementally during compilation. Each interesting unit of information is represented by a dependency node. Whenever a portion of the compilation process uses a piece of information that could change, the compiler adds an edge to the dependency graph from the node representing the information to the node representing the client of the information. When changes are made to the source program, the compiler computes what source dependency nodes have been affected and propagates invalidations downstream from these nodes. This invalidates all information (including compiled units) that depended on the changed source information. After processing all invalidations, invalidated compiled units are automatically recompiled.

To support greater invalidation selectivity, an internal node in the dependency framework can be a *filtering node*. When invalidated, a filtering node will first check whether the information it represents really has changed; only if the information it represents has changed will a filtering node invalidate its successor dependency nodes. For example, a filtering node could be encapsulating the information that a particular procedure may modify a specific global variable. When the procedure body is edited, the filtering node is invalidated, but the invalidation is only propagated to the node's dependents if the procedure no longer modifies the global variable. A common use of filtering nodes is to represent caches of information, such as interprocedural summaries.

5.1.2 Source Code to Call Graph

The program call graph is an abstraction of the calling relationship among a program's procedures, and thus will clearly be dependent on at least a subset of the program's source units. Exactly which source units the call graph depends on is a function of the call graph construction algorithm. For example, the call graph built by a simple algorithm such as $G_{selector}$ will only depend on the procedure declarations and call sites contained in the program; other aspects of the program's data or control flow are irrelevant. On the other hand, more sophisticated algorithms that conservatively approximate the program's dataflow to more precisely determine the set of receiver classes at message send sites will introduce complex and subtle dependencies from program source units to the program call graph. Explicitly representing and maintaining these dependencies is likely to incur prohibitively large time and space costs. In the worst-case, each call graph edge may depend on every other call graph node and edge; even under less pessimistic assumptions the representation and maintenance costs of an explicit dependency structure for the call graph could be quite large. Therefore, I propose an implicit representation of the dependencies from source units to call graph components.

The key idea of this approach is that for each kind of source code change, it is possible to conservatively approximate the set of impacted call graph nodes without maintaining any explicit dependency links from source units to call graph components. Incremental call graph construction

Table 5.1: Source code changes and the program call graph

Source Change	Direct Impact on Call Graph
A method body changes	The method's call graph node is marked out-of-date.
A method is added	A new call graph node is created (and is initially marked out-of-date). The method's generic function has been modified, therefore all call graph nodes containing call sites of the generic function are marked out-of-date.
A method is deleted	The call graph node is marked as deleted. The method's generic function has been modified, therefore all call graph nodes containing call sites of the generic function are marked out-of-date.
A class is added	No impact. Unless the class is instantiated, inheritance links are changed, or methods are added/deleted, the call graph is not impacted.
A class is deleted	No impact. Any changes will be due to deletion of inheritance links, method deletions and additions, and removal of instantiations from method bodies.
An inheritance link, **`class A inherits B`**, is added or deleted	The only possible impact of this source modification is to change the results of compile-time method lookup for some of the program's generic functions. The impacted generic functions are exactly those that contain methods that specialize on **`B`** or an ancestor of **`B`** (other than the root object). Any call graph node that contains a call site of one of these generic functions is marked out-of-date.
An instance variable declaration is added or deleted	No impact. Any changes will be due to modifications of method bodies to contain load/store sites of the instance variable.
A global variable declaration is added or deleted	No impact. Any changes will be due to modifications of method bodies to add/remove references to the variable.

proceeds in two phases. First, based on the source code changes, a set of directly out-of-date call graph nodes are identified. Second, interprocedural class analysis is incrementally performed starting from the previous call graph and class sets and the list of directly out-of-date nodes. This second phase has two effects: it produces a new sound call graph and class sets and it identifies all modified (out-of-date) and deleted call graph nodes. The rest of this section discusses these two phases in turn.

The exact details of the first phase, mapping from source code changes to the set of directly modified call graph nodes, depends on the constructs and semantics of the source language. Table 5.1 lists some examples of possible source program changes and their direct impact on the program call graph for the source language defined by figure 3.7. For simplicity, we assume that at each incremental reconstruction of the call graph, the source program is correct (contains no references to undefined variables, classes, instance variables, or generic functions). This restriction could be relaxed by slightly complicating some of the rules.[1]

The rules of table 5.1 require three mappings from source declarations to call graph nodes. Finding the call graph node that corresponds to a source method (or closure) is trivial in Vortex; its call graph data structure is a hash table-based keyed set that uses method (and closure) declarations as keys. Vortex must also be able to find all call sites of a given generic function. One possible implementation simply performs a linear scan over the program text to find the desired call sites. A second, more time efficient implementation, maintains for each generic function a set of its call sites. The space costs of these lists will be linear in the size of the program, and the lists can be easily updated as methods are added and deleted from the program. Finally, updating the call graph after inheritance links are modified requires a mapping from a class to all of its ancestor classes and from a class to all of the methods that are specialized on it. The first mapping is already trivially supported by Vortex's representation of the program's inheritance graph. Mapping from a class to the methods that specialize on it can be implemented in linear space by simply maintaining a set of specializing methods for each class in the program. These sets are updated as methods are added and deleted from the program.

Given the set of directly out-of-date call graph nodes, incremental interprocedural class analysis can proceed by first finding all of the contours that represent the directly out-of-date call graph nodes. These directly out-of-date contours are put on the algorithm's worklist and interprocedural class analysis is performed until a new fix-point is reached. As contours are reanalyzed, their call graph nodes are marked out-of-date, thus identifying all indirectly modified call graph nodes. All directly deleted call graph nodes were identified as methods were deleted,

---

1. For example, suppose that the input program was allowed to contain references to undeclared global variables during call graph construction. Then the addition/deletion of a global variable declaration would require all call graph nodes containing references to the variable to be explicitly marked out-of-date. This could be implemented either by scanning the program to find all methods that refer to the global variable or by maintaining on an explicit mapping of global variables to dependent call graph nodes. Similar solutions are required if a program can contain references to undeclared instance variables or classes. Cecil does allow programs to contain such references as long as they are not actually evaluated at runtime, so the actual Vortex implementation of incremental call graph construction would be required to handle these cases.

however call graph nodes that were included in the previous version of the call graph may no longer be reachable in the new call graph because all of their callers have been deleted. Incremental call graph construction could also mark these indirectly dead nodes as deleted if it maintained a list of nodes that correspond to methods in the program that currently have no callers; any node remaining on this list after the call graph is rebuilt could be marked as deleted. The sets of out-of-date and deleted call graph nodes are used to enable the incremental reconstruction of the various interprocedural summaries that depend on the call graph.

To make incremental interprocedural class analysis possible, in addition to preserving the call graph nodes, call graph edges, and class sets, Vortex must also either preserve or reconstruct the inter-contour dependencies induced by caller contours using the value returned by their callees, and by reads of global variables, instance variables, and variables declared in lexically enclosing scopes (as these class sets change, dependent contours must be identified and invalidated). The representation of these inter-contour dependencies is slightly different depending on whether the call graph construction algorithm uses an implicit or explicit representation of the program's dataflow graph. In both cases the current Vortex implementation chooses to persistently represent these inter-contour dependencies, but lazily recomputing some of them to reduce space costs may be a viable design alternative.

Because incremental call graph construction does not use $G_{\top}$ as its initial call graph, it is possible for it to arrive at a more pessimistic (lower in the call graph lattice) solution for a given program than would be computed if the same algorithm were run from scratch. As the program evolves, the difference in precision between the incremental and non-incremental call graphs will increase. It is unclear how to determine when it would be desirable to abandon an incrementally constructed call graph and build a new call graph from scratch.

A more precise call graph could be built (at a potentially higher cost in reanalysis time) if the *CallSite*, *LoadSite*, and *StoreSite* mappings of all out-of-date contours were cleared before incremental reanalysis was performed. In the absence of unification, all class contour sets of local variables could be cleared as well. New up-to-date versions of these mappings will be reconstructed by the reanalysis of the contour. The class contour sets corresponding to the contour's formal parameters could also be cleared and recomputed by analyzing all of the contour's callers to discover what class contours were passed as actual parameters. If the call graph construction algorithm generated equality or bounded inclusion constraints that were satisfied via unification of class sets, then it is not correct to clear a class set that had been unified with other class sets. The problem is that a class set “owned” by an out-of-date contour may have been unified with a class set “owned” by a contour that is neither directly nor indirectly out-of-date. If the class set is simply cleared, then it may not be correctly reconstructed due to missing contributions from non-modified contours.

It is possible to use *exact unions* to improve the precision of an incrementally constructed call graph. After the call graph is rebuilt, portions of the dataflow graph are examined to find class sets that contain elements that are not contained in any of the class sets immediately upstream of them in the dataflow graph. These extra element classes can be safely removed because the relevant inclusion constraints will still be satisfied. Initially, all class sets corresponding to formal parameters of out-of-date contours and all class sets on the frontier between out-of-date contours and up to date contours are candidates for improvement via exact unions. After an element is removed from a class set, all of its immediately downstream class sets should be examined as well. This process continues iteratively until either a fix-point is reached, or too much time has been consumed (it is always safe to stop because the current call graph and class sets are guaranteed to be sound).

A final algorithm-specific implementation detail is the design of space-efficient persistent representations for the contour call graph, class sets, and inter-contour dataflow dependencies. In most cases, the current internal representation should be adequate for an initial implementation. However, the p-Bounded Linear Edge algorithm uses a space-intensive circular linked list to represent bags of classes so that unification of two bags can be performed in constant time. Therefore, a post-pass over the dataflow graph to remove duplicate elements from the bags and/or convert them into a more compact bit set representation might result in substantial space savings. A simple hybrid scheme could use the final cardinality of a class set to choose between the two options for its persistent representation.

5.1.3 Call Graph to Interprocedural Summaries

Interprocedural analyses compute their results by traversing edges in the program call graph and analyzing the program’s procedures (represented by call graph nodes). Therefore all interprocedural summaries will depend on the call graph nodes and edges over which the interprocedural analysis was performed. The relationship between an element in an interprocedural summary and the call graph nodes and edges on which it depends can be quite complex, and thus is likely to be prohibitively expensive to represent explicitly in the dependency graph (or even to compute). Therefore, I propose that all dependencies between the call graph and interprocedural summaries be represented implicitly.

As part of incremental call graph construction, call graph nodes are marked as modified, unchanged, or deleted. This information is sufficient to enable interprocedural summaries to be updated after call graph modifications with only two small changes in the current Vortex interprocedural analysis framework:

- When the framework is invoked by a client, analysis always begins at the roots of the call graph. For incremental reanalysis, the client also needs to be able to specify that analysis should begin at an arbitrary collection of call graph nodes, namely those nodes that incremental call graph construction has marked as out-of-date.
- The framework relies on an internal, currently temporary, data structure to record inter-call graph node dependencies to ensure that nodes are reanalyzed as the client analysis converges to a fix-point and the information they depend on changes. For correct reanalysis, the information contained in this data structure needs to be persistent. It may be possible to avoid making this internal data structure persistent for context-insensitive interprocedural analyses, since the information is also implicitly recorded in the edges of the call graph, but it is unclear what the time/space trade-offs are.

After these two changes are implemented, updating interprocedural summaries after call graph changes is a simple matter of invoking each interprocedural analysis on the set of modified call graph nodes. However, further tuning may be desirable to find appropriate time/space trade-offs for each analysis. In particular, for some analyses, developing compact versions of the summaries may be necessary.

5.1.4 Interprocedural Summaries to Compiled Code

Dependencies are introduced between an interprocedural summary and a compiled unit whenever information encapsulated by the summary is exploited during the optimization of code contained in the compiled unit. The granularity at which clients can access, and thus depend on, interprocedural summaries is a crucial design decision, since it can greatly impact both the selectivity and cost of the dependencies. Because this design decision is highly dependent on analysis-specific details of what kinds of information are being exported to clients and how they exploit it, it is infeasible to describe an exact implementation of these dependencies for all interprocedural analyses. However, it is expected that most designs will have the following structure. An interprocedural summary contains internal state (the internal form of the analysis results) and a mapping from the internal state to externally accessible units of information. Each accessible unit is represented by a node in the dependency graph. Every time the compiler uses an accessible unit of information, it adds an edge in the dependency graph from the node representing the accessible unit to the node representing the client that exploited the information (for example, the compiled unit that was optimized). When the internal state of the summary changes (due to reanalysis triggered by cascading invalidations from source code changes), invalidation messages must be sent to the nodes representing all of the impacted accessible units of information.

As a concrete example, consider the two following designs for adding dependency nodes to the summary computed by interprocedural constant propagation. In both cases, the internal state of a constant propagation summary is a mapping of program variables to either "not a constant" or a particular constant value. In the first design, the accessible unit of information is fine-grained: each dependency node represents the value of a single variable. The fine-grained approach is highly selective, but may have a large space cost. A second possible design trades reduced selectivity for lower space costs by using a coarser-grained notion of an accessible unit of information: for example, the values of all variables declared in a single source unit (a file, module, or procedure). The benefit of grouping variables into larger accessible units is a reduction in the number of dependency nodes and edges required to support clients of interprocedural constant propagation. The cost is a reduction in selectivity, i.e., a change in the value of one variable in a unit invalidates all clients that depend on any variable in the unit. In practice, the coarse-grained design might not be significantly less selective, because it might be reasonable to expect that code that depends on one variable is also likely to depend on other variables declared in the same source unit.

5.1.5 Current Implementation Status

The primitive dependency mechanisms described in section 5.1.1 are implemented and have been an integral part of the Vortex compiler and programming environment for over four years. However, very little work has been done as yet to integrate call graph construction and call-graph based interprocedural analysis into the incremental recompilation framework. All of the experimental results reported in this dissertation were gathered by a system that contains no support for incremental interprocedural analysis.

Recently, members of the Cecil/Vortex project have performed a redesign and restructuring of Vortex's interprocedural analysis framework that should make it easier to implement the ideas described in sections 5.1.3 and 5.1.4. We expect that the remaining modifications to the interprocedural analysis framework required to support incremental analysis will be straightforward. The two most challenging aspects of the implementation will be efficiently supporting incremental call graph construction (section 5.1.2) and determining in practice what the most effective unit of accessible information is for each individual interprocedural analysis.

5.2 Related Work

Vortex's basic dependency mechanisms are closely related to those developed in the Self system [Chambers et al. 89]. Both systems use a fine-grained, persistent dependency graph to link compiled units to aspects of the program source code on which their compilation and optimization depended. There are a number of minor differences between the two systems due to details of the languages compiled and the optimizations performed by each system, but the only significant difference between the two systems is the addition in Vortex of filtering nodes. As discussed in more detail elsewhere, filtering nodes can increase selectivity by allowing a procedural check to be performed before propagating an invalidation to downstream nodes in the dependency graph [Chambers et al. 95]. This basic mechanism can be used to implement a variety of recompilation policies such as smart recompilation [Tichy & Baker 85], attribute recompilation [Dausmann 85] and the recompilation tests proposed by Burke and Torczon [Burke & Torczon 93].

The ParaScope programming environment was designed to support effective interprocedural analysis and optimization of Fortran programs. Part of its design was the development of a compilation model that included a persistent program database that contains interprocedural summaries and dependency information linking summaries to compiled units. A number of recompilation tests were developed to reduce the number of compiled units that were invalidated after program changes and thus must be recompiled [Cooper et al. 86, Burke & Torczon 93]. In contrast to the system proposed for Vortex, ParaScope's call graph construction and interprocedural analyses were always redone from scratch after the program's source code changed. After this batch recomputation of all interprocedural information, the new summaries were compared to the old summaries and any compiled code that depended on an invalidated aspect of a summary was recompiled. The FIAT system improves on the basic ParaScope approach by allowing interprocedural summaries to be computed in a demand-driven fashion [Hall et al. 93].

5.3 Summary

This chapter reviewed the existing dependency framework used by Vortex to support incremental recompilation in the presence of complex inter-compilation unit dependencies. Using the primitive mechanisms supplied by this framework, a design for supporting incremental call graph construction and interprocedural analysis was proposed. Initial implementation work is currently under way as part of the ongoing research effort of the Cecil/Vortex project.

Chapter 6

Conclusions

6.1 Results of this Work

A large body of prior work has investigated the problem of call graph construction for object-oriented and functional languages. The majority of prior work has focused on algorithm design: typically a single new algorithm is proposed and then proven correct and/or empirically evaluated. In contrast, algorithm design is only a minor contribution of this dissertation (I developed the SCS algorithm of section 4.4.1.2 and collaborated in the design of the *p*-Bounded and *p*-Bounded Linear Edge algorithms of section 4.5). The main contribution of this dissertation is the breadth with which it examines the call graph construction problem:

- A general framework for understanding the call graph construction problem is developed that formally defines both call graphs and a general parameterized call graph construction algorithm for a simple core language containing message sends and first class function values. The general algorithm facilitates the understanding of particular call graph construction algorithms by factoring out their common core, thus highlighting the few places where they can fundamentally differ: their contour key selection functions and their use of equality, bounded inclusion or inclusion constraints. A wide range of specific call graph construction algorithms, both well-known and novel, can be defined as particular instances of the general algorithm.
- Using the uniform vocabulary of the general call graph construction algorithm, a half dozen different specific algorithms that cover a wide spectrum of the algorithm design space are precisely and succinctly described. The algorithms range from the fast but relatively

imprecise *p*-Bounded Linear Edge family of algorithms to aggressive context-sensitive algorithms such as CPA and SCS.

- These call graph construction algorithms are empirically assessed on a suite of sizeable (5,000 to 50,000 lines) object-oriented programs. Data is presented on call graph construction costs, call graph precision, and the impact of call graph precision on the bottom-line effectiveness of a suite of interprocedural analyses. This experimental assessment is quite extensive; over 800 combinations of call graph construction algorithm, benchmark program, and other compiler optimizations are considered. Additionally, the benchmark suite includes programs that are an order of magnitude larger than those used in previous assessments of call graph construction algorithms; including larger programs is important, because many of the call graph construction algorithms have problems scaling to programs of more than a few thousand line of code. The results of these experiments demonstrate that interprocedural analysis can be both effectively and practically applied to sizeable object-oriented programs. For many of the benchmark applications, interprocedural analysis enabled substantial speed-ups over an already highly optimized baseline. Furthermore, a significant fraction of these speed-ups can be obtained through the use of a scalable, near-linear time call graph construction algorithm.
- Finally, some of the challenges of supporting interprocedural analysis in a programming environment used for normal day-to-day program development are discussed. A design for integrating Vortex's call graph construction and interprocedural analysis frameworks with its existing dependency framework is proposed, thus enabling incremental recompilation and interprocedural analysis to coexist.

6.2 Future Directions

This section highlights three areas of future research that I think are particularly promising. Sections 3.6 and 4.7 also discussed these ideas, along with some others.

6.2.1 Algorithm Design

Although some known algorithms are both scalable and effective, the search for the "optimal" call graph construction algorithm is still an open problem. This problem is especially challenging for sizeable programs: large purely object-oriented programs typically contain interesting uses of polymorphism that can only be accurately modeled by context-sensitive call graph construction algorithms, but the sheer size of these programs also make it impractical to apply any known context-sensitive call graph construction algorithm. One promising solution to this dilemma is to develop hybrid algorithms that utilize fast but imprecise algorithms to analyze large portions of the program and selectively increase context-sensitivity in portions of the program that are both polymorphic and important to analyze precisely. There are a number of possible approaches; the main design questions are how to most effectively blend analyses of varying precision, what context-sensitivity strategy to use, and how to determine which portions of the program merit more precise analysis. Plevyak's iterative algorithm [Plevyak & Chien 94] represents one possible approach for defining such an algorithm. However, although it may be successful in some languages, our experimental results suggest that Plevyak's iterative algorithm will not scale to large Cecil or Smalltalk programs. A more promising approach might be to combine context-sensitive analysis with bounded inclusion constraints. One such algorithm is Ashley's sub-1CFA algorithm that combines 1-CFA with bounded inclusion constraints [Ashley 97]. A second possible strategy of expending more analysis effort in frequently executed portions of the program is intriguing, but may be non-trivial to implement; the precise analysis of a program hot spot may depend on ensuring that the data structures it manipulates are precisely analyzed, which in turn may require introducing additional context-sensitivity in other parts of the program that manipulate these data structures and along any dataflow paths that connect these related portions of the program. In general, very little work has been done to design intelligent context-sensitivity strategies for data structures. Doing a better job of analyzing polymorphic container classes, such as lists, hash tables, and sets, is especially important for large programs, because they often contain many distinct clients of these basic classes, each of which uses them differently. Current algorithms either ignore the issue of polymorphic container classes or take a coarse-grained approach of applying the same context-sensitivity strategy to all of a container class's instance variables and methods. In languages with parameterized static type systems, type declarations could be used to identify which instance variables and methods need to be specialized and which could be analyzed in a context-insensitive fashion without any loss of precision.

6.2.2 Vortex Implementation

There are a number of improvements that could be made to the Vortex implementation of the call graph construction and interprocedural analysis frameworks. Three of the most interesting are exploring time/space trade-offs, incorporating on-line cycle elimination, and fully integrating interprocedural analysis into Vortex's incremental recompilation framework.

Finding the right balance of time/space trade-offs is an important part of both algorithm design and implementation. Some options were explored in the design of the Vortex implementation, but a number of additional experiments are possible. For example, using an explicit representation of the intra-contour dataflow graph in frequently reanalyzed contours and an implicit representation in the rest of the contours could reduce analysis time for context-sensitive algorithms without imposing the space overhead of explicitly representing the entire context-sensitive dataflow graph.

Recent work on on-line cycle elimination in inclusion constraint graphs has demonstrated that orders of magnitude reductions in analysis time can be obtained in at least some circumstances [Fähndrich et al. 98]. This work was done in the context of alias analysis, but it would be reasonable to expect similar results if the same techniques were applied to the inclusion constraint graphs generated during call graph construction. It is unclear how these techniques would interact with the unification mechanism currently used by Vortex to satisfy equality and bounded inclusion constraints.

Finally, a full-fledged implementation of the design for incremental recompilation proposed in chapter 5 could enable interprocedural analysis to be used in day-to-day Vortex compilation. It would also allow an exploration of the design space of the incremental mechanisms to see which time/space trade-offs matter in practice. Ideally, a study using traces of several months of actual program development, analogous to the one performed to evaluate the incremental recompilation mechanisms developed to support inlining and class hierarchy analysis [Chambers et al. 95], would be performed to validate the design.

6.2.3 Analyzing Incomplete Programs

Relaxing the "closed world" assumption, i.e. that all of an application's source code is available to be analyzed, would broaden the applicability of interprocedural analysis. Although the trend towards integrated development environments makes it increasingly likely that the entire program source will be available for analysis, there are common situations in which it will not be. For example, many applications utilize commercial class libraries that typically only provide header files and pre-compiled object files. There are two obvious solutions to this problem: either the missing portions of the program can be provided in an alternative non-source format that still enables precise analysis, such as Java bytecodes, summaries provided along with the libraries, or a textual form of the compiler's intermediate language, or the analysis must conservatively approximate the behavior of the missing portions of the program. In a type-safe language, static type information could be exploited to conservatively approximate the behavior of missing parts of the program while still enabling precise analysis of the available portions; this idea has been explored in the context of control flow analysis [Tang & Jouvelot 94, Debbabi et al. 96, Banerjee 97] and alias analysis [Diwan et al. 98].

6.3 Applicability of this Work

This dissertation assessed call graph construction algorithms in the context of supporting effective interprocedural analysis in an optimizing compiler for object-oriented languages. The analysis framework is directly applicable to functional languages, although it unclear to what degree the empirical results on the relative costs and benefits of particular call graph construction algorithms will be applicable for a functional language. One important aspect of the empirical studies is relevant both to the call graph construction problem for object-oriented and functional languages and more generally to the empirical assessment of any interprocedural analysis: it is important to evaluate algorithms on large programs, doing so may lead to quite different conclusions than if only small programs are considered. Finally, call graphs can also be utilized by a variety of other common programming environment tools. The experimental data on call graph construction costs and abstract precision could be used to guide the selection of the appropriate call graph construction algorithm(s) for inclusion in such tools.

Bibliography

[Agesen 94] Ole Agesen. Constraint-Based Type Inference and Parametric Polymorphism. In *First International Static Analysis Symposium*, September 1994.

[Agesen 95] Ole Agesen. The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism. In *Proceedings ECOOP '95*, Aarhus, Denmark, August 1995. Springer-Verlag.

[Agesen 96] Ole Agesen. *Concrete Type Inference: Delivering Object-Oriented Applications*. PhD thesis, Stanford, January 1996. SLMI TR 96-52.

[Agesen et al. 93] Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach. Type Inference of Self: Analysis of Objects with Dynamic and Multiple Inheritance. In O. Nierstrasz, editor, *Proceedings ECOOP '93*, LNCS 707, pages 247–267, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[Aho et al. 86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[Aiken 94] A. Aiken. Set Constraints: Results, Applications, and Future Directions. In *Second Workshop on the Principles and Practice of Constraint Programming*, pages 171–179, Orcas Island, Washington, May 1994.

[Allen 74] F. E. Allen. Interprocedural data flow analysis. In *Proceedings of the IFIP Congress*, pages 398–402, 1974.

[Alt & Martin 95] Martin Alt and Florian Martin. Generation of efficient interprocedural analyzers with PAG. In *Proceedings of the Second International Symposium on Static Analysis*, pages 33–50. Springer-Verlag, September 1995.

[Andersen 94] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994. DIKU report 94/19.

[Ashley 96] J. Michael Ashley. A Practical and Flexible Flow Analysis for Higher-Order Languages. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 184–194, January 1996.

[Ashley 97] J. Michael Ashley. The Effectiveness of Flow Analysis for Inlining. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 99–111, Amsterdam, The Netherlands, June 1997.

[Bacon & Sweeney 96] David F. Bacon and Peter F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *OOPSLA'96 Conference Proceedings*, San Jose, CA, October 1996.

[Bairagi et al. 97] Deepankar Bairagi, Sandeep Kumar, and Dharma P. Agrawal. Precise Call Graph Construction for OO Programs in the Presence of Virtual Functions. In *The 1997 International Conference on Parallel Processing*, pages 412–416, August 1997.

[Banerjee 97] Anindya Banerjee. A Modular, Polyvariant, and Type-Based Closure Analysis. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 1–10, Amsterdam, The Netherlands, June 1997.

[Bieman & Zhao 95] James M. Bieman and Josephine Xia Zhao. Reuse Through Inheritance: A Quantitative Study of C++ Software. In *Proceedings of the Symposium on Software Reusability.* ACM SIGSOFT, August 1995. Software Engineering Notes.

[Bobrow et al. 88] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common Lisp Object System Specification X3J13. *SIGPLAN Notices*, 28(Special Issue), September 1988.

[Burke & Torczon 93] Michael Burke and Linda Torczon. Interprocedural Optimization: Eliminating Unnecessary Recompilation. *ACM Transactions on Programming Languages and Systems*, 15(3):367–399, July 1993.

[Calder & Grunwald 94] Brad Calder and Dirk Grunwald. Reducing Indirect Function Call Overhead in C++ Programs. In *Conference Record of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 397–408, January 1994.

[Callahan 88] David Callahan. The Program Summary Graph and Flow-Sensitive Interprocedural Data Flow Analysis. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 47–56, June 1988..

[Callahan et al. 86] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural Constant Propagation. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 152–161, July 1986.

[Callahan et al. 90] David Callahan, Alan Carle, Mary W. Hall, and Ken Kennedy. Constructing the Procedure Call Multigraph. *IEEE Transactions on Software Engineering*, 16(4):483–487, 1990.

[Chambers & Ungar 89] Craig Chambers and David Ungar. Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 146–160, June 1989.

[Chambers & Ungar 90] Craig Chambers and David Ungar. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 150–164, June 1990.

[Chambers 93] Craig Chambers. The Cecil Language: Specification and Rationale. Technical Report UW-CSE-93-03-05, Department of Computer Science and Engineering. University of Washington, March 1993.

[Chambers et al. 89] Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of Self – a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *Proceedings OOPSLA '89*, pages 49–70, October 1989. Published as ACM SIGPLAN Notices, volume 24, number 10.

[Chambers et al. 95] Craig Chambers, Jeffrey Dean, and David Grove. A Framework for Selective Recompilation in the Presence of Complex Intermodule Dependencies. In *17th International Conference on Software Engineering*, Seattle, WA, April 1995.

[Chambers et al. 96a] Craig Chambers, Jeffrey Dean, and David Grove. Frameworks for Intra- and Interprocedural Dataflow Analysis. Technical Report UW-CSE-96-11-02, University of Washington, November 1996.

[Chambers et al. 96b] Craig Chambers, Jeffrey Dean, and David Grove. Whole-Program Optimization of Object-Oriented Languages. Technical Report UW-CSE-96-06-02, Department of Computer Science and Engineering. University of Washington, June 1996.

[Cooper et al. 86] Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural Optimization: Eliminating Unnecessary Recompilation. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 58–67, July 1986.

[Cooper et al. 92] Keith D. Cooper, Mary W. Hall, and Ken Kennedy. Procedure Cloning. In *Proceedings of 1992 IEEE International Conference on Computer Languages*, pages 96–105, Oakland, CA, April 1992.

[Cousot & Cousot 77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

[Dausmann 85] M. Dausmann. Informationstrukuren und Verfahren fur die getrennte Ubersetzung von Programmteilen. Technical report, GMD-Bericht 155, R. Oldenburg Verlag, Munich, Germany, 1985.

[Dean 96] Jeffrey Dean. *Whole Program Optimization of Object-Oriented Languages*. PhD thesis, University of Washington, November 1996. TR-96-11-05.

[Dean et al. 95a] Jeffrey Dean, Craig Chambers, and David Grove. Selective Specialization for Object-Oriented Languages. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 93–102, June 1995.

[Dean et al. 95b] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings ECOOP '95*, Aarhus, Denmark, August 1995. Springer-Verlag.

[Dean et al. 96] Jeffrey Dean, Greg DeFouw, Dave Grove, Vassily Litvinov, and Craig Chambers. Vortex: An Optimizing Compiler for Object-Oriented Languages. In *OOPSLA'96 Conference Proceedings*, San Jose, CA, October 1996.

[Debbabi et al. 96] Mourad Debbabi, Ali Faour, and Nadia Tawbi. A Type-Based Algorithm for the Control-Flow Analysis of Higher-Order Concurrent Programs. In *IFL'96: Implementation of Functional Languages*, LNCS 1268, pages 247–266. Springer-Verlag, September 1996.

[DeFouw et al. 98] Greg DeFouw, David Grove, and Craig Chambers. Fast Interprocedural Class Analysis. In *Conference Record of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 222–236, January 1998.

[Deutsch & Schiffman 84] L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 297–302, January 1984.

[Diwan et al. 96] Amer Diwan, Eliot Moss, and Kathryn McKinley. Simple and Effective Analysis of Statically-typed Object-Oriented Programs. In *OOPSLA'96 Conference Proceedings*, San Jose, CA, October 1996.

[Diwan et al. 98] Amer Diwan, Kathryn McKinley, and J. Eliot B. Moss. Type-Based Alias Analysis. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 106–117, June 1998.

[Dolby 97] Julian Dolby. Automatic Inline Allocation of Objects. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 7–17, June 1997.

[Emami et al. 94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.

[Fähndrich et al. 98] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial Online Cycle Elimination in Inclusion Constraint Graphs. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 85–96, June 1998.

[Fernandez 95] Mary F. Fernandez. Simple and Effective Link-Time Optimization of Modula-3 Programs. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 103–115, June 1995.

[Goldberg & Robson 83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addision-Wesley, Reading, MA, 1983.

[Gosling et al. 96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, Reading, MA, 1996.

[Grove & Torczon 93] Dan Grove and Linda Torczon. Interprocedural Constant Propagation: A Study of Jump Function Implementations. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 90–99, June 1993.

[Grove 95] David Grove. The Impact of Interprocedural Class Analysis on Optimization. In *Proceedings CASCON '95*, pages 195–203, Toronto, Canada, October 1995.

[Grove et al. 95] David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers. Profile-Guided Receiver Class Prediction. In *OOPSLA'95 Conference Proceedings*, pages 108–123, Austin, TX, October 1995.

[Grove et al. 97] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call Graph Construction in Object Oriented Languages. In *OOPSLA'97 Conference Proceedings*, Atlanta, GA, October 1997.

[Hall & Kennedy 92] Mary W. Hall and Ken Kennedy. Efficient Call Graph Analysis. *ACM Letters on Programming Languages and Systems*, 1(3):227–242, September 1992.

[Hall et al. 93] M.W. Hall, J.M. Mellor-Crummey, A. Carle, and R. Rodriguez. FIAT: A Framework for Interprocedural Analysis and Transformation. In *The Sixth Anunual Workshop on Parallel Languages and Compilers*, August 1993.

[Heintze & McAllester 97] Nevin Heintze and David McAllester. Linear-Time Subtransitive Control Flow Analysis. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 261–272, June 1997.

[Henglein 91] Fritz Henglein. Efficient Type Inference for Higher-Order Binding-Time Analysis. In *Functional Programming and Computer Architecture*, 1991.

[Hölzle & Agesen 96] Urs Hölzle and Ole Agesen. Dynamic vs. Static Optimization Techniques for Object-Oriented Languages. *Theory and Practice of Object Systems*, 1(3), 1996.

[Hölzle & Ungar 94] Urs Hölzle and David Ungar. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 326–336, June 1994.

[Jagannathan & Weeks 95] Suresh Jagannathan and Stephen Weeks. A Unified Treatment of Flow Analysis in Higher-Order Languages. In *Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 393–407, January 1995.

[Johnson et al. 88] Ralph E. Johnson, Justin O. Graver, and Lawrence W. Zurawski. TS: An Optimizing Compiler for Smalltalk. In *Proceedings OOPSLA '88*, pages 18–26, November 1988. Published as ACM SIGPLAN Notices, volume 23, number 11.

[Jouppi 90] Norm Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *The 17th International Symposium on Computer Architecture*, pages 364–373, Seattle, Washington, May 1990.

[Kam & Ullman 76] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, January 1976.

[Kildall 73] Gary A. Kildall. A Unified Approach to Global Program Optimization. In *Conference Record of the First ACM Symposium on Principles of Programming Languages*, pages 194–206, October 1973.

[Kranz 88] David Kranz. *ORBIT: An Optimizing Compiler for Scheme*. PhD thesis, Yale University, February 1988. Department of Computer Science, Research Report 632.

[Lakhotia 93] Arun Lakhotia. Constructing Call Multigraphs Using Dependence Graphs. In *Conference Record of the Twentieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 273–284, January 1993.

[Landi & Ryder 91] William Landi and Barbara G. Ryder. Pointer-Induced Aliasing: A Problem Classification. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 93–103, January 1991.

[Larchevêque 94] J.M. Larchevêque. Interprocedural Type Propagation for Object-Oriented Languages. *Science of Computer Programming*, 22(3):257–282, June 1994.

[Lea 90] Doug Lea. Customization in C++. In *Proceedings of the 1990 Usenix C++ Conference*, San Francisco, CA, April 1990.

[Marlowe et al. 95] Thomas J. Marlowe, Barbara G. Ryder, and Michael G. Burke. Defining Flow Sensitivity in Data Flow Problems. Technical Report LCSR-249, Rutgers University, July 1995.

[Milner et al. 90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[Muchnick 97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA, 1997.

[Nielson & Nielson 97] Flemming Nielson and Hanne Riis Nielson. Infinitary Control Flow Analysis: A Collecting Semantics for Closure Analysis. In *Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 332–345, January 1997.

[Odersky & Wadler 97] Martin Odersky and Philip Wadler. Pizza into Java: Translating Theory into Practice. In *Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 146–159, January 1997.

[Oxhøj et al. 92] Nicholas Oxhøj, Jens Palsberg, and Michael I. Schwartzbach. Making Type Inference Practical. In O. Lehrmann Madsen, editor, *Proceedings ECOOP '92*, LNCS 615, pages 329–349, Utrecht, The Netherlands, June 1992. Springer-Verlag.

[Palsberg & Pavlopoulou 98] Jens Palsberg and Christina Pavlopoulou. From Polyvariant Flow Information to Intersection and Union Types. In *Conference Record of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 197–208, January 1998.

[Palsberg & Schwartzbach 91] Jens Palsberg and Michael I. Schwartzbach. Object-Oriented Type Inference. In *Proceedings OOPSLA '91*, pages 146–161, November 1991. Published as ACM SIGPLAN Notices, volume 26, number 11.

[Pande & Ryder 94] Hemant D. Pande and Barbara G. Ryder. Static Type Determination for C++. In *Proceedings of Sixth USENIX C++ Technical Conference*, 1994.

[Phillips & Shepard 94] G. Phillips and T. Shepard. Static Typing Without Explicit Types. Unpublished report, Dept. of Electrical and Computer Engineering, Royal Military College of Canada, Kingston, Ontario, Canada, 1994.

[Plevyak & Chien 94] John Plevyak and Andrew A. Chien. Precise Concrete Type Inference for Object-Oriented Languages. In *Proceedings OOPSLA '94*, pages 324–340, Portland, OR, October 1994.

[Plevyak & Chien 95] John Plevyak and Andrew A. Chien. Type Directed Cloning for Object-Oriented Programs. In *Proceedings of the Workshop for Languages and Compilers for Parallel Computing*, pages 566–580, 1995.

[Plevyak 96] John Plevyak. *Optimization of Object-Oriented and Concurrent Programs*. PhD thesis, University of Illinois at Urbana-Champaign, 1996.

[Rees & Clinger 86] Jonathan Rees and William Clinger. Revised^3 Report on the Algorithmic Language Scheme. *SIGPLAN Notices*, 21(12), December 1986.

[Ryder 79] Barbara Ryder. Constructing the Call Graph of a Program. *IEEE Transactions on Software Engineering*, 5(3):216–225, 1979.

[Shalit 96] Andrew Shalit, editor. *The Dylan Reference Manual*. Addison-Wesley, Reading, MA, 1996.

[Shapiro & Horwitz 97] Marc Shapiro and Susan Horwitz. Fast and Accurate Flow-Insensitive Points-To Analysis. In *Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, January 1997.

[Sharir & Pnueli 81] Micha Sharir and Amir Pnueli. Two Approaches to Interprocedural Data Flow Analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–233. Prentice-hall, 1981.

[Shivers 88] Olin Shivers. Control-Flow Analysis in Scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 164–174, June 1988.

[Shivers 91] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, May 1991. CMU-CS-91-145.

[Spillman 71] T. C. Spillman. Exposing side-effects in a PL/1 optimizing compiler. In *Proceedings of the IFIP Congress*, pages 376–381, 1971.

[Steensgaard 94] Bjarne Steensgaard. A Polyvariant Closure Analysis with Dynamic Abstraction. Unpublished manuscript, 1994.

[Steensgaard 96] Bjarne Steensgaard. Points-to Analysis in Almost Linear Time. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, January 1996.

[Stefanescu & Zhou 94] Dan Stefanescu and Yuli Zhou. An Equational Framework for the Flow Analysis of Higher-Order Functional Programs. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 190–198, June 1994.

[Stroustrup 91] Bjarne Stroustrup. *The C++ Programming Language (second edition)*. Addision-Wesley, Reading, MA, 1991.

[Tang & Jouvelot 94] Yan Mei Tang and Pierre Jouvelot. Separate Abstract Interpretation for Control-Flow Analysis. In *TACS'94: Theoretical Aspects of Computer Software*, LNCS 789, pages 224–243. Springer-Verlag, April 1994.

[Tarjan 75] Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[Tichy & Baker 85] Walter F. Tichy and Mark C. Baker. Smart Recompilation. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 236–244, January 1985.

[Ungar & Smith 87] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *Proceedings OOPSLA '87*, pages 227–242, December 1987.

[Vitek et al. 92] J. Vitek, N. Horspool, and J.S. Uhl. Compile Time Analysis of Object-Oriented Programs. In *Proceedings of the CC'92. 4'th International Conference on Compiler Construction*, pages 237–250. Springer-Verlag, October 1992.

[Weihl 80] William E. Weihl. Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables and Label Variables. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 83–94, January 1980.

[Weiser 84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[Wilson & Lam 95] Robert P. Wilson and Monica S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1–12, June 1995.

[Wright & Jagannathan 98] Andrew K. Wright and Suresh Jagannathan. Polymorphic splitting: an effective polyvariant flow analysis. *ACM Transactions on Programming Languages and Systems*, 20(1):166–207, January 1998.

Appendix A

Detailed Description of Benchmark Programs

To understand why (or why not) interprocedural analyses improve application performance and to understand the relative performance of the various call graph construction algorithms, it is helpful to have more detailed information about the program. A program's structure and the degree to which it uses object-oriented language features can have a profound impact on the difficulty of the call graph construction problem and on the performance impact of interprocedural analysis. Therefore, this appendix presents data that characterizes the "object-orientedness" of the applications and the frequency of use of those language features that complicate call graph construction.

The following metrics characterize the basic structure of the application:

- *Lines of Code:* Total lines of code (including comments and white space).
- *Number of Classes*: The number of classes declared in the program.
- *Number of Dispatched Methods*: The number of methods declared in the program that have at least one non-trivial specializer, i.e. are not simply global procedures.
- *Number of Immediate Children*: Measures the number of classes that directly inherit from each class in the program; indicates the branching factor (breadth) and "bushiness" of the class hierarchies.
- *Maximum Distance to Root of Inheritance Hierarchy*: The longest path from each class to the root of its inheritance hierarchy; indicates the depth of the class hierarchies used by the program.

Bieman and Zhao used the last two of these metrics (and some additional ones) in their study of code reuse through inheritance in C++ applications [Bieman & Zhao 95].

The following additional metrics characterize the inherent difficulty of the call graph construction problem for the application:

- *Number of Callee Methods*: Measures the cardinality of the callee set, i.e. the number of possible targets (dispatched methods, procedures and/or closures), at every call site in a call graph, optionally weighted by the execution frequency of the call site. Numbers are reported for two call graphs, $G_{selector}$ and G_{prof}, thus suggesting a lower and upper bound on the static and dynamic number of callee methods at a typical call site.
- *Number of Closures*: Number of closures declared in the program; indicates how commonly function values are created.
- *Number of Call Sites*: Total number of call sites in the program
- *Percentage of Message Send Sites*: The percentage of all program call sites that are dynamically dispatched message sends.
- *Percentage of Closure Application Sites*: The percentage of all program call sites that are potentially applications of closure values.

Many of the benchmark programs were also described using a subset of these metrics and some additional metrics elsewhere [Dean et al. 96, Dean 96].

Table A.1: Detailed description of benchmark programs

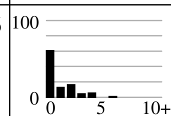
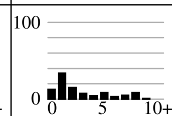
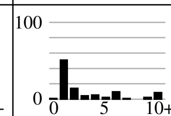
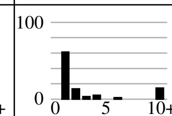
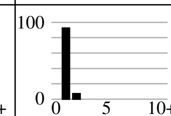
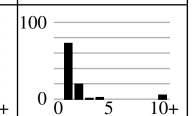
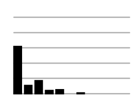
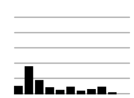
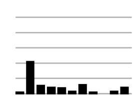
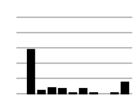
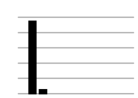
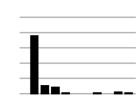
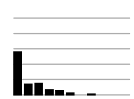
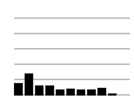
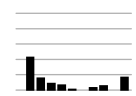
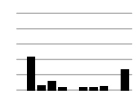
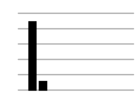
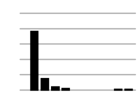
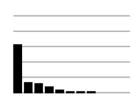
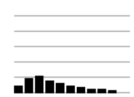
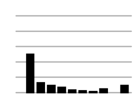
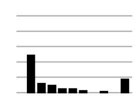
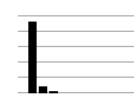
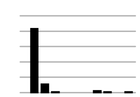
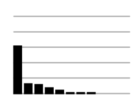
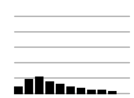
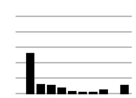
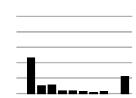
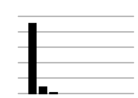
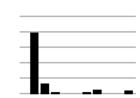
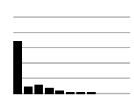
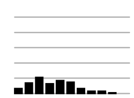
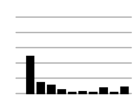
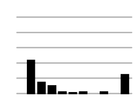
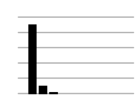
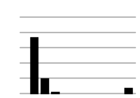
	Program	Lines of Code	# of Classes	# of Dispatched Methods	# of Closures	# of Call Sites	% Message Send Sites	% Closure Application Sites	% of classes with *x* immediate children (x = 0..9,10+)	% of classes with distance *x* from root of hierarchy (x = 0..9,10+)	% of call sites with *x* callee methods (x = 0..9,10+)			
											$G_{selector}$		G_{prof}	
											Static	Dynamic	Static	Dynamic
Cecil	richards	400 + 4,850 std. library	107	641	482	2,133	86	6						
	deltablue	650 + 4,850 std. library	106	625	517	2214	88	6						
	instr sched	2,400 + 10,700 std. library	236	1,648	1,513	7,219	91	4						
	type-checker	20,000 + 10,700 std. library	635	4,924	3,918	19,537	90	2						
	new-tc	23,500 + 10,700 std. library	624	4,940	4,316	20,706	89	2						
	compiler	50,000 + 10,700 std. library	1,197	8,368	7,431	38,764	90	1						

Table A.1: Detailed description of benchmark programs

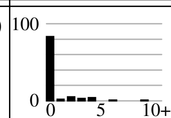
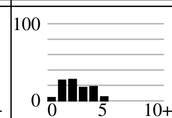
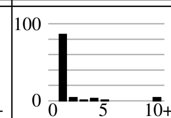
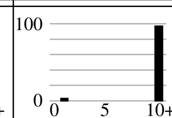
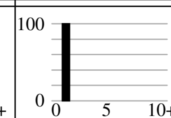
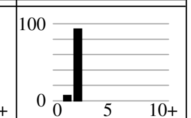
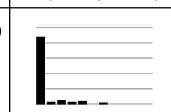
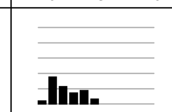
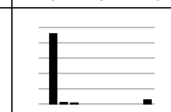
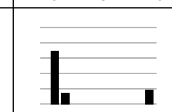
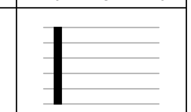
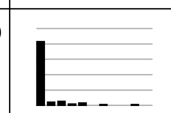
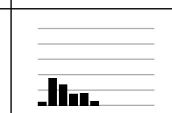
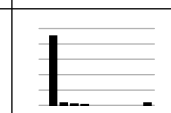
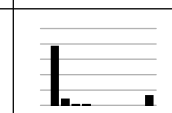
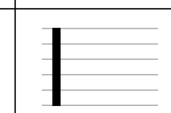
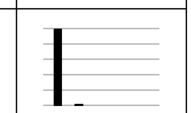
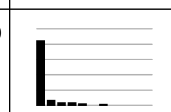
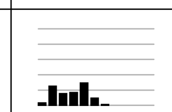
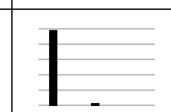
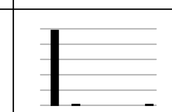
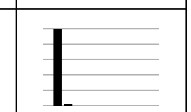
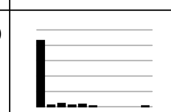
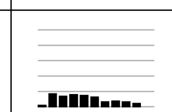
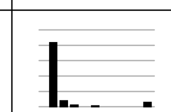
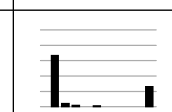
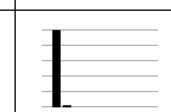
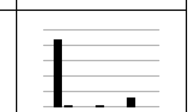
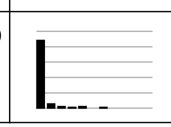
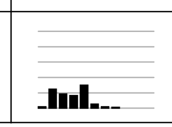
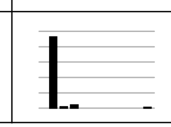
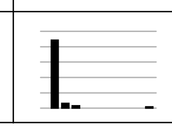
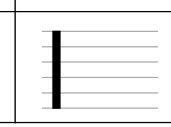
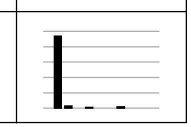
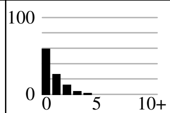
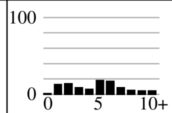
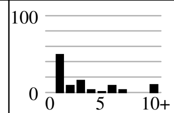
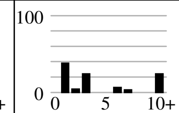
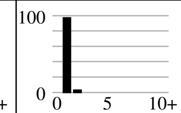
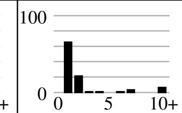
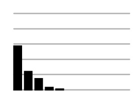
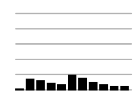
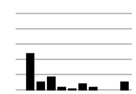
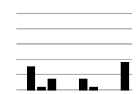
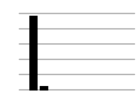
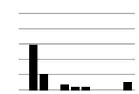
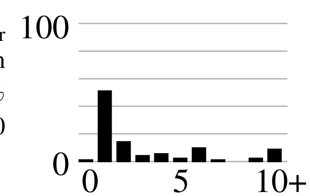
	Program	Lines of Code	# of Classes	# of Dispatched Methods	# of Closures	# of Call Sites	% Message Send Sites	% Closure Application Sites	% of classes with *x* immediate children (x = 0..9,10+)	% of classes with distance *x* from root of hierarchy (x = 0..9,10+)	% of call sites with *x* callee methods (x = 0..9,10+)			
											$G_{selector}$		G_{prof}	
											Static	Dynamic	Static	Dynamic
Java	cas-sowary	3,400 + 16,400 std. library	155	1,839	0	6,164	46	0						
	toba	3,900 + 16,400 std. library	146	1,559	0	8,413	48	0						
	java-cup	7,800 + 16,400 std. library	163	1,994	0	9,269	48	0						
	espresso	13,800 + 16,400 std. library	232	2,496	0	13,552	28	0						
	javac	25,500 + 16,400 std. library	297	3,615	0	17,287	49	0						
	pizza	27,500 + 16,400 std. library	335	3,848	0	23,844	32	0						

Table A.1: Detailed description of benchmark programs

	Program	Lines of Code	# of Classes	# of Dispatched Methods	# of Closures	# of Call Sites	% Message Send Sites	% Closure Application Sites	% of classes with *x* immediate children (x = 0..9,10+)	% of classes with distance *x* from root of hierarchy (x = 0..9,10+)	% of call sites with *x* callee methods (x = 0..9,10+) $G_{selector}$ Static	$G_{selector}$ Dynamic	G_{prof} Static	G_{prof} Dynamic
Smalltalk	richards	695 + std. library	253	5,293	2,569	12,390	83	1						
	deltablue	1,154 + std. library	257	5,339	2,623	12,596	83	1						

To the right is an enlarged copy of the histogram displaying the static number of message sends with *x* callee methods in $G_{selector}$ for the Cecil version of the richards benchmark. This histogram shows that roughly 50% of the messages sent by the program are sent at call sites where the selector based call graph construction has determined there is only 1 possible callee method, 15% are sent from call sites determined to have 2 callee methods, and 10% are sent from call sites that were determined to have 10 or more callee methods.



Appendix B

Experimental Data

This appendix contains the raw data for the experiments described in chapter 4. Section B.1 reports on the costs of call graph construction and interprocedural analysis and on the abstract precision of the resulting call graphs. Section B.2 reports the impact on application performance. Some of the metrics are discussed in more detail in section 4.1.3.

B.1 Analysis Cost and Precision

Analysis time cost is measured using the following metrics:

- *Call graph construction time*: The combined system and user CPU time (in seconds) required to build the call graph on a Sun Ultra 1 Model 170 SparcStation with 384 MB of physical memory and 2.3 GB of virtual memory running Solaris 5.5.1.
- *Heap growth*: The increase in program heap size (in megabytes) observed during call graph construction.
- *Interprocedural analysis time*: The combined system and user CPU time (in seconds) required to perform each interprocedural analysis. Note that times are not reported for tree-shaking or class analysis because these analyses are performed as part of call graph construction.

As discussed in section 4.1.3, all call graph precision data is for the summarized procedure call graphs derived from the context-sensitive contour call graphs, because it is the procedure call graphs that are exported to the rest of Vortex for use during interprocedural analysis. The following metrics are used to report call graph precision:

- *Percent Reachable Methods*: What fraction of the program's methods are included in the call graph, i.e., are reachable from the main routine? In addition to measuring call graph precision, this metric also suggests how effective the treeshaking optimization will be at reducing executable size.
- *Average Number of Callees*: What is the average number of callee procedures at a call site?
- *Percent Singleton Callees*: What percentage of the call sites in the call graph have exactly one callee? These call sites are likely candidates for static binding and inlining.

Table B.1: Analysis time costs and call graph precision

	CGC Algorithm	CGC Time	Interprocedural Analysis Times				Heap Growth	% Reachable Methods	Average Callees	% Singleton Callees
			MOD	Escape	Except	Total				
richards (Cecil)	selector	2.30	2.58	12.04	1.27	15.90	2	62	6.78	48
	0-Bounded LE	3.29	2.65	11.46	1.35	15.46	2	54	6.14	55
	2-Bounded LE	3.52	2.57	11.76	1.53	15.85	2	53	6.11	55
	4-Bounded LE	3.35	2.39	11.05	1.28	14.73	2	51	6.06	54
	8-Bounded LE	3.56	2.10	11.10	1.46	14.66	2	51	6.06	54
	16-Bounded LE	4.19	2.21	10.85	1.19	14.25	4.50	51	6.06	54
	32-Bounded LE	3.89	1.99	10.81	1.38	14.18	2	46	5.64	54
	64-Bounded LE	3.63	2.11	10.73	1.16	13.99	2	46	5.64	54
	∞-Bounded LE	3.65	2.21	10.64	1.27	14.12	2	46	5.64	54
	0-Bounded	4.40	6.78	13.26	1.72	21.75	2	54	6.28	56
	2-Bounded	4.66	4.24	12.44	1.70	18.38	2	53	5.91	57
	4-Bounded	4.43	3.27	12.42	1.61	17.31	2	51	5.79	57
	8-Bounded	4.67	3.23	11.87	1.50	16.61	2	51	5.77	57
	16-Bounded	5.51	3.01	11.74	1.62	16.37	2	51	5.77	57
	32-Bounded	4.46	2.44	11.25	1.38	15.08	2	46	5.23	59
	64-Bounded	4.54	2.29	11.05	1.48	14.83	4.50	46	5.23	59
	∞-Bounded	4.66	2.59	11.22	1.37	15.18	2	46	5.23	59
	0CFA	3.52	0.84	5.60	0.59	7.02	1	27	1.19	89
	1-0CFA	4.47	0.95	5.54	0.62	7.11	2	27	1.19	89
	2-0CFA	4.74	0.99	5.53	0.59	7.11	2	27	1.19	89
	3-0CFA	5.65	0.77	5.55	0.89	7.21	2	27	1.19	89
	1-1CFA	5.22	0.71	5.98	0.89	7.58	2	27	1.19	89
	2-2CFA	5.18	0.78	5.45	0.70	6.93	2	27	1.19	89
	3-3CFA	5.27	0.89	5.41	0.81	7.12	2	27	1.19	89
	bounded-CPA	4.33	0.91	5.22	0.49	6.62	1	25	1.19	90
	SCS	3.68	0.71	5.36	0.54	6.61	2	25	1.19	90
	prof(optimstic)	4.76	0.75	5.36	0.49	6.60	2	24	1.16	92
deltablue (Cecil)	selector	2.29	2.73	19.81	1.53	24.08	2	64	9.68	40
	0-Bounded LE	3.36	2.17	18.67	1.74	22.59	2	56	8.96	43
	2-Bounded LE	3.46	2.21	19.09	1.77	23.07	2	54	8.56	44
	4-Bounded LE	3.48	2.18	18.99	1.54	22.71	2	52	8.50	44
	8-Bounded LE	3.55	2.20	18.76	1.55	22.50	2	52	8.50	44
	16-Bounded LE	4.78	2.05	18.51	1.57	22.13	4.50	52	8.50	44
	32-Bounded LE	3.88	2.11	18.55	1.51	22.17	2	50	8.30	44
	64-Bounded LE	3.67	2.12	18.26	1.40	21.77	2	50	8.30	44
	∞-Bounded LE	3.93	1.95	18.34	1.52	21.82	2	50	8.30	44
	0-Bounded	4.99	4.29	21.54	2.67	28.50	2	56	9.20	45
	2-Bounded	5.45	3.31	22.49	2.04	27.85	2	54	8.26	49
	4-Bounded	5.32	3.15	21.44	2.12	26.72	2	52	8.08	50
	8-Bounded	5.35	2.99	21.76	1.93	26.68	2	52	8.05	50
	16-Bounded	5.87	3.09	21.31	1.92	26.32	4.50	52	8.05	50
	32-Bounded	5.64	3.31	20.30	1.80	25.41	4.40	50	7.65	50
	64-Bounded	5.70	3.55	20.36	1.77	25.68	2	50	7.65	50
	∞-Bounded	5.66	3.80	19.36	1.90	25.06	4.40	50	7.65	50
	0CFA	5.49	1.64	11.47	1.16	14.28	2	31	1.41	79
	1-0CFA	7.06	1.51	11.50	1.21	14.21	2	31	1.41	79
	2-0CFA	8.46	1.68	11.13	1.15	13.96	2	30	1.40	80
	3-0CFA	11.06	1.63	11.34	1.15	14.13	2	30	1.39	80
	1-1CFA	7.14	1.37	12.54	1.20	15.11	2	30	1.39	80
	2-2CFA	8.86	1.59	11.20	1.03	13.82	2	30	1.37	80
	3-3CFA	11.61	1.65	11.03	1.11	13.78	2	30	1.37	81
	bounded-CPA	10.91	1.39	11.13	1.31	13.82	4.40	30	1.40	80
	SCS	8.70	1.61	11.23	1.21	14.04	2	30	1.40	80
	prof(optimstic)	5.65	1.09	7.70	0.80	9.59	2	27	1.20	92

Table B.1: Analysis time costs and call graph precision

	CGC Algorithm	CGC Time	Interprocedural Analysis Times				Heap Growth	% Reachable Methods	Average Callees	% Singleton Callees
			MOD	Escape	Except	Total				
instr sched (Cecil)	selector	5.10	10.08	101.25	7.22	118.55	3.40	82	21.80	39
	0-Bounded LE	5.93	7.40	72.75	4.75	84.91	2.50	54	15.18	41
	2-Bounded LE	6.32	7.34	70.93	4.47	82.74	5.80	54	15.05	41
	4-Bounded LE	6.24	7.11	73.22	4.30	84.64	5.80	54	14.23	41
	8-Bounded LE	6.28	7.28	72.95	4.32	84.55	3.30	54	14.27	41
	16-Bounded LE	7.94	6.83	73.11	4.50	84.44	7.20	54	14.27	41
	32-Bounded LE	17.85	7.22	70.42	4.44	82.09	15.30	54	14.27	41
	64-Bounded LE	8.86	5.81	69.02	4.25	79.08	9.90	45	12.44	42
	∞-Bounded LE	8.47	6.33	68.88	4.89	80.11	10	45	12.44	42
	0-Bounded	16.55	22.74	81.95	11.66	116.35	7.20	54	15.18	41
	2-Bounded	17.69	21.55	84.44	8.52	114.51	9.70	54	13.47	45
	4-Bounded	17.88	21.38	84.76	8.92	115.06	7.50	54	13.10	46
	8-Bounded	17.57	20.17	86.83	8.73	115.74	7.50	54	13.02	46
	16-Bounded	19.22	20.19	85.72	7.95	113.86	12.30	54	13.01	46
	32-Bounded	21.61	20.87	70.09	5.93	96.90	19.00	45	10.91	51
	64-Bounded	21.76	16.40	70.58	8.59	95.57	21.40	45	10.90	51
	∞-Bounded	21.79	16.17	70.42	7.88	94.47	21.40	45	10.90	51
	0CFA	63.30	6.42	50.85	3.85	61.12	5.80	37	1.68	69
	1-0CFA	93.00	6.47	49.15	4.01	59.63	9.70	37	1.62	72
	2-0CFA	144.34	5.19	47.74	4.01	56.94	12.30	36	1.55	76
	3-0CFA	326.80	6.02	47.40	3.41	56.84	20.90	36	1.55	76
	1-1CFA	118.04	6.49	51.86	3.62	61.96	14.80	37	1.61	72
	2-2CFA	370.71	5.46	48.83	3.73	58.02	14.80	36	1.54	76
	3-3CFA	1,838.33	5.74	46.36	3.27	55.37	26.50	36	1.53	76
	bounded-CPA	203.19	5.39	46.78	3.84	56.01	12.30	36	1.53	76
	SCS	119.83	5.26	44.52	3.78	53.55	14.50	36	1.54	76
	prof(optimstic)	14.27	3.06	22.68	1.84	27.58	3.60	28	1.30	87
typechecker (Cecil)	selector	16.77	39.99	400.58	28.75	469.32	0	89	43.52	44
	0-Bounded LE	23.52	33.40	377.69	23.20	434.28	6.70	82	39.85	47
	2-Bounded LE	25.54	33.86	382.62	24.72	441.20	15.00	81	39.73	47
	4-Bounded LE	25.84	31.49	383.80	26.29	441.58	12.10	81	39.72	47
	8-Bounded LE	25.08	31.08	379.66	26.41	437.15	20.20	81	39.71	47
	16-Bounded LE	27.06	33.75	380.88	26.36	440.98	20.20	81	39.71	47
	32-Bounded LE	36.32	34.40	374.60	24.35	433.36	26	81	39.71	47
	64-Bounded LE	32.90	34.19	381.66	26.22	442.07	30.90	81	39.71	47
	∞-Bounded LE	130.02	27.61	366.45	21.50	415.56	99.20	75	36.78	49
	0-Bounded	157.88	301.00	675.06	212.77	1,188.83	38.20	82	39.88	47
	2-Bounded	166.26	185.11	546.57	122.41	854.08	54.90	81	37.35	53
	4-Bounded	167.10	179.63	581.97	117.78	879.37	54.80	81	37.12	53
	8-Bounded	167.04	181.38	584.36	120.55	886.28	72.00	81	36.98	53
	16-Bounded	167.25	189.55	589.20	119.70	898.44	71.80	81	36.98	53
	32-Bounded	169.78	182.08	589.85	120.88	892.81	71.90	81	36.98	53
	64-Bounded	202.73	185.32	593.04	125.15	903.52	91.50	81	36.98	53
	∞-Bounded	630.90	79.46	639.47	60.06	779.00	419.80	74	31.63	66
	0CFA	1,052.32	374.59	349.69	221.56	945.84	60.30	83	4.20	72
	1-0CFA	24,261.18	137.62	394.81	81.84	614.27	107.50	72	2.96	74
	2-0CFA									
	3-0CFA									
	1-1CFA									
	2-2CFA									
	3-3CFA									
	bounded-CPA									
	SCS									
	prof(optimstic)	87.53	26.98	197.47	19.13	243.57	22.20	49	1.30	89

Table B.1: Analysis time costs and call graph precision

	CGC Algorithm	CGC Time	Interprocedural Analysis Times				Heap Growth	% Reachable Methods	Average Callees	% Singleton Callees
			MOD	Escape	Except	Total				
new-tc (Cecil)	selector	17.83	47.86	466.39	32.31	546.57	8.10	90	52.71	45
	0-Bounded LE	24.46	40.16	427.68	28.97	496.81	7.10	83	49.09	49
	2-Bounded LE	26.59	39.10	419.09	29.60	487.80	15.30	82	49.07	49
	4-Bounded LE	27.14	37.91	426.29	30.93	495.13	15.30	82	49.06	49
	8-Bounded LE	26.31	38.10	418.49	29.64	486.23	15.30	82	49.06	49
	16-Bounded LE	29.20	37.84	421.59	29.76	489.19	15.50	82	49.06	49
	32-Bounded LE	41.27	38.04	424.32	27.39	489.75	26.40	82	49.06	49
	64-Bounded LE	46.66	37.91	425.25	28.68	491.83	40.50	82	49.06	49
	∞-Bounded LE	144.20	35.33	404.81	20.72	460.87	99.40	76	46.41	51
	0-Bounded	157.48	319.51	633.99	264.61	1,218.11	39.50	83	49.47	49
	2-Bounded	166.06	502.41	589.03	138.63	1,230.07	72.90	82	47.11	54
	4-Bounded	163.22	174.28	699.96	114.57	988.81	55.50	82	46.87	55
	8-Bounded	164.33	181.99	692.96	122.15	997.09	72.80	82	46.74	55
	16-Bounded	166.52	170.57	688.95	113.97	973.50	73	82	46.74	55
	32-Bounded	167.72	173.19	692.06	114.69	979.93	73	82	46.74	55
	64-Bounded	197.17	179.11	708.18	117.86	1,005.15	94.30	82	46.74	55
	∞-Bounded	574.57	581.83	979.70	212.82	1,774.35	412	75	41.98	65
	0CFA	1,113.13	99.01	457.42	86.38	642.81	56.20	83	3.90	72
	1-0CFA	22,865.51	57.61	432.01	44.01	533.63	108.60	73	2.78	74
	2-0CFA									
	3-0CFA									
	1-1CFA									
	2-2CFA									
	3-3CFA									
	bounded-CPA									
	SCS									
	prof(optimstic)	94.69	31.10	238.32	21.65	291.08	28	51	1.31	89
compiler (Cecil)	selector	36.78	117.73	1,218.95	91.59	1,428.27	13.10	102	67.13	42
	0-Bounded LE	52.66	117.37	1,230.56	91.59	1,439.52	27.60	98	64.02	44
	2-Bounded LE	54.65	119.48	1,229.74	92.73	1,441.96	27.60	98	64	44
	4-Bounded LE	55.04	115.49	1,233.28	91.51	1,440.27	37.10	98	64	44
	8-Bounded LE	55.57	118.59	1,238.45	93.76	1,450.80	37.10	98	64	44
	16-Bounded LE	56.82	114.99	1,236.24	92.00	1,443.23	37.10	98	64	44
	32-Bounded LE	62.77	117.05	1,232.65	95.39	1,445.09	37.60	98	64	44
	64-Bounded LE	72.53	120.84	1,248.08	92.50	1,461.42	55.50	98	64	44
	∞-Bounded LE	717.60	121.91	1,403.79	100.34	1,626.03	402.70	95	61.51	45
	0-Bounded	723.30	974.74	2,104.21	803.60	3,882.55	124.90	98	64.02	44
	2-Bounded	778.47	671.75	2,062.75	545.42	3,279.91	211.90	98	60.43	50
	4-Bounded	794.39	680.26	2,061.18	563.27	3,304.71	202.60	98	60.08	50
	8-Bounded	790.82	667.40	2,042.47	555.99	3,265.87	211.60	98	60.01	50
	16-Bounded	808.40	627.10	2,017.68	525.08	3,169.86	219.70	98	60	50
	32-Bounded	775.73	674.10	2,046.40	574.52	3,295.03	229.50	98	60	50
	64-Bounded	816.16	637.18	2,080.03	544.63	3,261.85	230.50	98	60	50
	∞-Bounded									
	0CFA	11,781.03	4,115.32	1,512.99	2,154.92	7,783.23	201.70	99	9.30	68
	1-0CFA									
	2-0CFA									
	3-0CFA									
	1-1CFA									
	2-2CFA									
	3-3CFA									
	bounded-CPA									
	SCS									
	prof(optimstic)	280.49	102.88	781.48	86.44	970.79	72.80	66	1.24	87

Table B.1: Analysis time costs and call graph precision

	CGC Algorithm	CGC Time	Interprocedural Analysis Times				Heap Growth	% Reachable Methods	Average Callees	% Singleton Callees
			MOD	Escape	Except	Total				
cassowary (Java)	selector	19.53	8.46	57.36	7.07	72.90	3.10	27	2.67	82
	0-Bounded LE	18.18	6.27	44.49	6.23	56.99	12.60	21	2.02	82
	2-Bounded LE	19.40	6.68	43.83	6.18	56.70	7.20	21	2.02	82
	4-Bounded LE	19.20	6.02	44.81	5.76	56.60	7.50	21	2.02	82
	8-Bounded LE	19.04	6.40	43.40	6.09	55.89	7.20	21	2.02	82
	16-Bounded LE	19.60	6.24	44.30	6.21	56.75	7.20	21	2.01	82
	32-Bounded LE	18.81	6.41	42.64	5.59	54.64	7.20	20	1.80	82
	64-Bounded LE	18.92	5.85	43.16	5.65	54.67	7.50	20	1.80	82
	∞-Bounded LE	19.10	6.13	42.94	5.73	54.80	7.20	20	1.80	82
	0-Bounded	23.31	6.39	44.67	5.97	57.03	7.20	21	2.03	83
	2-Bounded	22.94	6.63	44.70	6.03	57.36	7.50	21	1.47	83
	4-Bounded	22.62	6.77	43.14	6.10	56.00	7.20	21	1.35	84
	8-Bounded	23.51	6.34	43.48	6.87	56.69	7.20	21	1.35	84
	16-Bounded	22.38	5.59	37.20	4.93	47.72	7.50	20	1.21	86
	32-Bounded	22.54	5.54	38.59	5.55	49.68	7.20	20	1.21	86
	64-Bounded	23.19	5.60	37.00	5.01	47.60	7.20	20	1.21	86
	∞-Bounded	23.02	5.53	38.10	4.89	48.51	7.30	20	1.21	86
	0CFA	21.44	5.50	38.07	4.92	48.49	6.90	21	1.11	98
	1-0CFA	27.40	4.77	35.45	4.79	45.01	7.20	19	1.05	98
	2-0CFA	40.57	4.99	35.93	4.47	45.40	6.90	19	1.05	98
	3-0CFA	69.94	4.91	36.05	4.98	45.95	7.50	19	1.05	98
	1-1CFA	77.48	4.36	34.13	4.51	43	7	18	1.03	98
	2-2CFA	647.23	4.77	33.55	4.36	42.68	12.30	18	1.03	98
	3-3CFA	12,033.01	4.64	32.92	3.94	41.51	18.50	18	1.03	98
	bounded-CPA	32.09	4.60	36.52	4.60	45.72	7	19	1.05	98
	SCS	26.18	5.06	35.40	4.85	45.31	6.90	19	1.05	98
	prof(optimstic)	22.37	4.84	32.67	4.50	42.01	7.40	17	1.02	99
toba (Java)	selector	40.64	13.33	260.64	11.86	285.84	10	30	2.12	90
	0-Bounded LE	43.22	11.60	280.07	10.93	302.59	16.10	26	1.72	90
	2-Bounded LE	44.50	11.89	287.11	11.13	310.14	10	26	1.72	90
	4-Bounded LE	43.61	12.42	289.14	12.33	313.90	15.50	26	1.72	90
	8-Bounded LE	44.30	11.85	288.09	11.46	311.40	16.20	26	1.72	90
	16-Bounded LE	45.33	12.85	249.37	12.39	274.61	15.30	26	1.72	90
	32-Bounded LE	45.75	13.03	253.03	12.15	278.21	17	26	1.72	90
	64-Bounded LE	47.08	12.80	241.64	12.16	266.60	16.10	25	1.70	90
	∞-Bounded LE	45.57	12.56	238.82	12.08	263.45	15.70	25	1.70	90
	0-Bounded	52.72	12.41	240.18	10.88	263.48	15.80	26	1.72	90
	2-Bounded	54.28	12.38	242.04	12.17	266.58	15.50	26	1.72	90
	4-Bounded	50.44	12.51	241.10	11.21	264.83	16.20	26	1.18	91
	8-Bounded	53.23	12.72	242.54	12.42	267.68	15.50	26	1.18	91
	16-Bounded	51.64	12.84	255.12	11.27	279.23	15.80	26	1.18	91
	32-Bounded	52.32	12.44	246.58	11.92	270.94	15.50	26	1.18	91
	64-Bounded	51.46	11.46	241.34	11.83	264.62	15.80	25	1.11	92
	∞-Bounded	53.08	13.45	254.59	12.81	280.85	16.10	25	1.11	92
	0CFA	49.70	12.21	238.93	9.97	261.12	8.90	26	1.05	99
	1-0CFA	55.95	11.10	237.49	10.67	259.26	9	24	1.03	99
	2-0CFA	65.76	10.46	243.39	11.06	264.92	15.20	24	1.03	99
	3-0CFA	80.56	11.00	243.47	11.16	265.63	16.20	24	1.03	99
	1-1CFA	247.28	11.28	232.13	10.75	254.15	15	24	1.02	99
	2-2CFA	1,445.85	11.20	245.39	10.12	266.71	15.30	24	1.01	99
	3-3CFA	7,105.43	10.84	244.68	10.53	266.05	15.50	24	1.01	99
	bounded-CPA	89.88	11.06	242.07	10.38	263.51	9.60	25	1.03	99
	SCS	56.46	10.89	243.08	9.93	263.90	8.90	25	1.03	99
	prof(optimstic)	69.12	10.41	231.76	9.74	251.92	25.30	21	1	100

Table B.1: Analysis time costs and call graph precision

	CGC Algorithm	CGC Time	Interprocedural Analysis Times				Heap Growth	% Reachable Methods	Average Callees	% Singleton Callees
			MOD	Escape	Except	Total				
java-cup (Java)	selector	45.48	13.33	276.17	12.41	301.91	17.10	32	2.99	86
	0-Bounded LE	46.97	12.02	266.06	12.24	290.32	23.80	29	2.61	86
	2-Bounded LE	47.20	13.22	264.84	11.96	290.01	23.30	29	2.61	86
	4-Bounded LE	49.24	13.12	262.77	11.55	287.44	24.40	29	2.61	86
	8-Bounded LE	49.85	14.06	271.13	12.48	297.67	23.10	29	2.61	86
	16-Bounded LE	50.41	13.43	262.42	12.18	288.03	24	29	2.61	86
	32-Bounded LE	48.96	12.99	264.69	11.83	289.51	23.90	27	2.22	87
	64-Bounded LE	50.88	13.29	267.24	11.78	292.31	23.60	27	2.22	87
	∞-Bounded LE	52.24	12.70	264.99	12.24	289.93	23.30	27	2.30	86
	0-Bounded	57.46	14.68	262.22	12.14	289.04	23.80	29	2.64	86
	2-Bounded	59.77	14.32	267.60	12.40	294.32	24.10	29	1.43	86
	4-Bounded	58.12	14.35	266.11	12.12	292.58	23.80	29	1.37	87
	8-Bounded	57.87	14.49	264.09	13.60	292.18	24	29	1.37	87
	16-Bounded	60.72	15.05	266.21	13.32	294.58	23.80	29	1.37	87
	32-Bounded	60.10	11.73	249.11	12.01	272.84	24.20	27	1.15	89
	64-Bounded	57.97	12.48	254.51	11.84	278.83	23.80	27	1.15	89
	∞-Bounded	59.34	13.13	260.34	11.12	284.60	23.50	27	1.15	89
	0CFA	51.71	10.54	248.79	9.97	269.30	23.50	27	1.06	99
	1-0CFA	62.22	11.09	244.72	9.30	265.10	22.80	25	1.03	99
	2-0CFA	71.20	11.60	246.91	10.17	268.68	22.80	25	1.03	99
	3-0CFA	87.53	11.62	245.18	9.94	266.74	23.90	25	1.03	99
	1-1CFA	449.17	10.92	244.73	9.12	264.78	23	25	1.02	99
	2-2CFA	1,787.02	10.40	241.55	10.07	262.01	22.80	25	1.02	99
	3-3CFA	10,255.42	11.04	244.03	10.47	265.54	23.50	25	1.02	99
	bounded-CPA	71.25	10.46	264.61	9.63	284.71	22.70	25	1.03	99
	SCS	56.37	10.38	251.81	9.74	271.93	23.40	25	1.03	99
	prof(optimstic)	58.72	8.40	229.06	7.41	244.88	25.90	21	1.01	99
espresso (java)	selector	68.35	29.95	228.14	23.91	282	6.20	47	2.11	90
	0-Bounded LE	75.72	30.81	210.61	23.97	265.39	19.90	44	1.98	90
	2-Bounded LE	76.82	30.86	210.98	25.25	267.08	19.50	44	1.97	91
	4-Bounded LE	77.60	31.82	214.16	25.50	271.48	19.50	43	1.94	91
	8-Bounded LE	76.85	29.66	211.95	24.49	266.11	19.40	43	1.94	91
	16-Bounded LE	78.30	29.31	211.45	25.68	266.45	20.20	43	1.94	91
	32-Bounded LE	79.38	29.88	211.35	26.05	267.28	20.20	43	1.94	91
	64-Bounded LE	81.42	30.51	208.78	25.45	264.75	29.90	43	1.94	91
	∞-Bounded LE	83.12	28.74	212.92	23.17	264.83	20	42	1.91	91
	0-Bounded	93.19	29.82	409.12	24.57	463.51	20.50	44	1.99	91
	2-Bounded	88.94	30.04	409.71	24.48	464.22	19.50	44	1.90	91
	4-Bounded	89.26	30.85	403.39	23.03	457.27	20.20	43	1.83	92
	8-Bounded	89.87	32.16	400.76	26.17	459.09	20.30	43	1.83	92
	16-Bounded	89.81	31.11	402.29	23.89	457.30	19.50	43	1.83	92
	32-Bounded	87.94	32.89	403.80	25.45	462.15	20	43	1.82	92
	64-Bounded	97.31	28.84	396.25	23.37	448.45	20.40	43	1.82	92
	∞-Bounded	100.26	28.43	357.53	24.00	409.96	20.30	42	1.66	92
	0CFA	87.43	27.56	365.43	22.83	415.81	11.70	42	1.62	95
	1-0CFA	560.37	28.53	355.87	22.39	406.79	19.60	41	1.61	95
	2-0CFA	8,149.09	28.36	362.43	21.37	412.16	145	41	1.60	95
	3-0CFA									
	1-1CFA	7,685.61	27.52	348.58	22.00	398.10	41.10	41	1.56	95
	2-2CFA									
	3-3CFA									
	bounded-CPA	230.69	29.11	358.33	24.53	411.96	19.70	41	1.60	95
	SCS	179.96	29.90	356.56	24.68	411.14	19.20	41	1.59	95
	prof(optimstic)	93.61	24.61	204.32	19.16	248.09	23	35	1.08	98

Table B.1: Analysis time costs and call graph precision

	CGC Algorithm	CGC Time	Interprocedural Analysis Times				Heap Growth	% Reachable Methods	Average Callees	% Singleton Callees
			MOD	Escape	Except	Total				
javac (Java)	selector	131.78	52.40	388.18	45.13	485.71	8.50	44	3.55	76
	0-Bounded LE	140.17	51.72	563.16	44.83	659.72	31.70	42	3.26	76
	2-Bounded LE	144.07	50.47	567.41	47.11	664.99	18.60	42	3.26	76
	4-Bounded LE	144.56	49.97	567.77	43.53	661.26	19.70	42	3.26	76
	8-Bounded LE	140.97	50.62	569.64	46.57	666.83	30.30	42	3.26	76
	16-Bounded LE	144.21	52.16	567.31	45.19	664.65	30.40	42	3.26	76
	32-Bounded LE	147.45	50.47	568.97	44.91	664.36	30.30	42	3.26	76
	64-Bounded LE	156.66	50.24	575.21	42.54	667.99	31.90	42	3.26	76
	∞-Bounded LE	160.69	48.42	558.81	42.40	649.62	28.20	41	3.18	76
	0-Bounded	180.75	49.78	814.07	38.98	902.83	28	42	3.27	76
	2-Bounded	185.57	53.16	683.54	43.49	780.20	28.40	42	3.19	76
	4-Bounded	198.72	48.93	678.88	41.05	768.86	27.90	42	3.10	77
	8-Bounded	182.82	53.22	681.95	42.30	777.48	27.90	42	3.09	77
	16-Bounded	185.32	50.34	679.40	42.19	771.93	28	42	3.09	77
	32-Bounded	188.02	51.40	676.54	40.33	768.27	28.30	42	3.09	77
	64-Bounded	195.25	51.86	682.21	41.52	775.59	29.50	42	3.09	77
	∞-Bounded	216.18	42.72	703.77	35.36	781.86	67.50	41	2.23	79
	0CFA	190.01	38.09	601.90	31.69	671.68	17.70	40	2.11	88
	1-0CFA	1,679.97	42.13	697.41	34.95	774.49	29.60	39	2.07	87
	2-0CFA									
	3-0CFA									
	1-1CFA	89,196.08	37.52	716.70	30.67	784.89	61.70	39	2.06	88
	2-2CFA									
	3-3CFA									
	bounded-CPA	1,083.68	40.23	602.90	31.98	675.11	29.40	40	2.11	88
	SCS	525.95	39.41	606.44	33.89	679.73	17.60	40	2.11	88
	prof(optimstic)	131.49	30.08	173.79	27.64	231.50	18.40	29	1.12	96
pizza (java)	selector	146.08	75.20	711.02	58.80	845.02	20.70	49	1.70	89
	0-Bounded LE	161.72	75.44	731.64	59.38	866.47	32	45	1.48	89
	2-Bounded LE	163.75	76.84	725.55	61.46	863.85	31.70	45	1.47	90
	4-Bounded LE	163.34	79.32	729.66	61.98	870.97	30.50	44	1.46	90
	8-Bounded LE	164.47	80.96	726.62	65.61	873.19	32	44	1.45	90
	16-Bounded LE	165.49	74.94	716.60	59.11	850.64	31.70	44	1.45	90
	32-Bounded LE	159.64	78.78	714.05	59.45	852.27	31.50	44	1.45	90
	64-Bounded LE	170.37	78.60	720.23	62.09	860.93	31.50	44	1.45	90
	∞-Bounded LE	192.83	76.41	723.28	58.29	857.98	32.30	43	1.42	90
	0-Bounded	188.66	83.58	736.40	61.48	881.47	32.80	45	1.48	90
	2-Bounded	197.34	81.77	723.11	61.27	866.15	32	45	1.42	90
	4-Bounded	189.78	77.75	716.49	62.04	856.28	31.80	44	1.36	90
	8-Bounded	197.33	78.41	704.20	58.54	841.15	31.40	44	1.35	91
	16-Bounded	193.66	82.18	722.45	61.27	865.91	32.40	44	1.35	91
	32-Bounded	189.26	82.15	724.22	59.35	865.72	31.50	44	1.34	91
	64-Bounded	192.67	81.35	721.86	63.59	866.80	31.70	44	1.34	91
	∞-Bounded	213.81	74.88	709.62	57.59	842.09	48.70	43	1.19	92
	0CFA	192.34	75.87	697	57.71	830.58	29.50	42	1.14	96
	1-0CFA	448.23	74.38	700.51	56.54	831.43	30.50	41	1.12	96
	2-0CFA	899.38	72.18	697.25	54.92	824.35	46.70	41	1.12	96
	3-0CFA	3,049.77	74.41	707.70	58.53	840.64	99.90	41	1.12	96
	1-1CFA	43,945.95	71.77	713.22	53.28	838.27	63.40	41	1.11	97
	2-2CFA									
	3-3CFA									
	bounded-CPA	1,363.12	74.79	715.53	57.32	847.65	46.40	42	1.12	96
	SCS	889.85	76.14	713.84	62.13	852.10	30.70	42	1.12	96
	prof(optimstic)	195.97	71.56	644.94	50.18	766.68	39.50	36	1.02	99

Table B.1: Analysis time costs and call graph precision

	CGC Algorithm	CGC Time	Interprocedural Analysis Times				Heap Growth	% Reachable Methods	Average Callees	% Singleton Callees
			MOD	Escape	Except	Total				
richards (Smalltalk)	selector	143.51	72.02	851.02	39.73	962.78	11.40	19	25.40	44
	0-Bounded LE	153.07	55.39	596.00	36.65	688.04	23	19	25.40	44
	2-Bounded LE	155.45	59.07	643.88	37.38	740.33	15.10	19	25.06	44
	4-Bounded LE	155.56	56.78	642.03	37.83	736.64	15	19	25.06	44
	8-Bounded LE	156.32	56.51	647.08	39.48	743.08	15	19	25.05	44
	16-Bounded LE	157.69	58.60	641.22	39.08	738.90	15.30	19	25.05	44
	32-Bounded LE	158.66	53.61	642.50	37.13	733.25	23.30	19	25.05	44
	64-Bounded LE	170.25	56.48	636.69	37.31	730.48	34.70	19	25.05	44
	∞-Bounded LE	211.58	57.19	601.11	34.55	692.85	38.10	18	24.48	45
	0-Bounded	206.40	160.41	737.07	35.57	933.06	17.20	19	25.44	44
	2-Bounded	200.71	53.34	696.04	38.04	787.42	25.10	19	24.82	46
	4-Bounded	201.47	50.04	643.25	35.99	729.28	23.40	19	24.87	46
	8-Bounded	201.37	54.22	643.02	37.33	734.58	23.40	19	24.87	46
	16-Bounded	204.60	53.48	647.93	36.38	737.79	25.40	19	24.87	46
	32-Bounded	209.69	53.60	642.41	36.05	732.06	25.50	19	24.87	46
	64-Bounded	225.36	48.17	645.25	36.31	729.73	40.60	19	24.86	46
	∞-Bounded	468.66	46.18	714.26	36.69	797.13	128.20	18	24.09	51
	0CFA	1,686.32	40.75	280.79	29.43	350.96	32.50	15	3.07	61
	1-0CFA									
	2-0CFA									
	3-0CFA									
	1-1CFA									
	2-2CFA									
	3-3CFA									
	bounded-CPA									
	SCS									
	prof(optimstic)	62.76	11.43	58.78	10.08	80.29	5.10	5	1.12	96
deltablue (Smalltalk)	selector	143.75	70.61	843.64	42.64	956.89	11.40	19	26.93	43
	0-Bounded LE	153.26	60.68	624.21	38.06	722.95	22.90	19	26.93	43
	2-Bounded LE	152.81	55.58	567.62	37.29	660.49	15	19	26.57	43
	4-Bounded LE	153.39	54.45	566.49	38.84	659.78	15	19	26.69	43
	8-Bounded LE	154.73	57.67	587.11	40.21	684.99	15.20	19	26.68	43
	16-Bounded LE	156.48	55.67	581.90	40.09	677.66	15.30	19	26.68	43
	32-Bounded LE	156.39	56.25	578.43	40.59	675.27	22.90	19	26.68	43
	64-Bounded LE	177.91	54.49	579.51	39.03	673.03	34.60	19	26.68	43
	∞-Bounded LE	230.68	54.94	569.96	34.20	659.10	40.10	18	26.17	44
	0-Bounded	203.23	160.28	741.88	37.20	939.37	16.70	19	27.08	43
	2-Bounded	204.50	53.74	646.68	34.81	735.22	25.10	19	26.45	44
	4-Bounded	203.20	52.23	634.32	37.74	724.29	25.10	19	26.50	44
	8-Bounded	201.01	54.51	643.00	37.03	734.54	25.60	19	26.49	44
	16-Bounded	201.65	53.97	640.96	36.48	731.41	25.50	19	26.49	44
	32-Bounded	214.03	54.49	645.71	35.70	735.90	25.90	19	26.49	44
	64-Bounded	215.77	53.44	648.98	35.84	738.25	40.30	19	26.49	44
	∞-Bounded	479.38	51.62	718.72	37.53	807.87	139.20	18	25.64	50
	0CFA	1,493.45	40.43	275.27	29.29	344.99	43.90	16	3.12	61
	1-0CFA									
	2-0CFA									
	3-0CFA									
	1-1CFA									
	2-2CFA									
	3-3CFA									
	bounded-CPA									
	SCS									
	prof(optimstic)	61.35	13.52	56.81	10.59	80.92	11.20	5	1.15	94

B.2 Impact on Application Performance

This section reports the impact of call graph precision and interprocedural analysis on bottom-line application performance. The following metrics are used:

- *Application execution time*: The combined system and user CPU time (in seconds) required to run the application.
- *Executable size*: The size (in kilobytes) of the text, data, and bss segments of the compiled application minus the fixed-size cost of the language-specific Vortex runtime system. The runtime system sizes are: Cecil, 566 KB, Java, 502 KB, Smalltalk, 512 KB.
- *Number of calls*: The dynamic number of statically bound procedure calls that occurred during application execution.
- *Number of message sends*: The dynamic number of message sends (including closure applications) that occurred during application execution.
- *Percentage of exception-possible calls*: The dynamic percentage of procedure calls and message sends from call sites at which Vortex had to assume that an exception might be raised by a callee procedure. This metric indicates the effectiveness of exception detection analysis.
- *Number of class tests*: The dynamic number of class and subclass tests that occurred during application execution.
- *Number of heap allocated closures and environments*: The dynamic number of closure and environment objects that were potentially escaping, and thus were heap allocated (instead of stack allocated). This number is artificially low for the Cecil programs, because Vortex allows programmers to annotate closures as non-escaping, and virtually all non-escaping closures in the benchmark programs are so-annotated. In previous work, we examined the effectiveness of escape analysis for an alternative **`base`** configuration in which Vortex ignored the source-level non-escaping annotations [Grove et al. 97]. We found that the non-annotated **`base`** configuration was 50% slower than the default **`base`**, but that augmenting the non-annotated **`base`** configuration with interprocedural escape analysis resulted in performance that was essentially the same as the default **`base`**. Thus, interprocedural escape

analysis was sufficient to enable virtually all of the closure optimizations enabled by the source-level non-escaping annotations. All of the Vortex configurations measured in this dissertation do exploit the non-escaping annotations to avoid overstating the benefits of interprocedural analysis (since almost all non-escaping closures in Cecil source code are annotated, Vortex does not include several intraprocedural optimizations that might significantly reduce the costs of assuming that all closures could escape).

Table B.2: Application performance

		CGC Algorithm	Execution Time	Code Size	# of Calls	# of Sends	% Except Possible	# of Class Tests	# of Heap Alloc
richards (Cecil)	*without* profile-guided receiver class prediction	none (base)	0.127	415	2,673,846	1,294,143	100.00	15,217,350	0
		selector	0.130	257	2,673,846	1,294,143	84.53	15,217,350	0
		0-Bounded LE	0.103	229	2,673,846	723,694	81.94	13,339,463	0
		2-Bounded LE	0.096	219	2,673,847	723,692	81.94	11,185,760	0
		4-Bounded LE	0.094	211	2,673,847	723,692	81.94	9,815,743	0
		8-Bounded LE	0.093	211	2,673,847	723,692	81.94	9,815,743	0
		16-Bounded LE	0.094	211	2,673,847	723,692	81.94	9,815,743	0
		32-Bounded LE	0.094	206	2,673,847	723,692	81.94	9,594,489	0
		64-Bounded LE	0.094	206	2,673,847	723,692	81.94	9,594,489	0
		∞-Bounded LE	0.094	206	2,673,847	723,692	81.94	9,594,489	0
		0-Bounded	0.102	229	2,673,846	723,694	81.94	13,339,463	0
		2-Bounded	0.096	219	2,673,847	723,692	81.94	11,185,760	0
		4-Bounded	0.093	209	2,673,847	723,692	81.94	9,815,743	0
		8-Bounded	0.093	209	2,673,847	723,692	81.94	9,815,743	0
		16-Bounded	0.093	209	2,673,847	723,692	81.94	9,815,743	0
		32-Bounded	0.092	204	2,673,847	723,692	81.94	9,594,489	0
		64-Bounded	0.094	204	2,673,847	723,692	81.94	9,594,489	0
		∞-Bounded	0.092	204	2,673,847	723,692	81.94	9,594,489	0
		0CFA	0.091	144	2,673,848	723,691	81.94	6,870,658	0
		1-0CFA	0.090	145	2,673,848	723,691	81.94	6,870,658	0
		2-0CFA	0.090	145	2,673,848	723,691	81.94	6,870,658	0
		3-0CFA	0.090	145	2,673,848	723,691	81.94	6,870,658	0
		1-1CFA	0.088	135	2,673,848	723,691	81.94	6,870,658	0
		2-2CFA	0.090	145	2,673,848	723,691	81.94	6,870,658	0
		3-3CFA	0.090	145	2,673,848	723,691	81.94	6,870,658	0
		bounded-CPA	0.086	140	2,673,848	723,691	81.94	6,870,658	0
		SCS	0.085	140	2,673,848	723,691	81.94	6,870,658	0
		prof (optimisitc)	0.077	138	2,673,849	723,690	81.94	4,485,893	0
	with profile-guided receiver class prediction	none (base)	0.107	421	2,673,846	723,826	100.00	14,486,708	0
		selector	0.102	260	2,673,846	723,826	81.93	14,486,708	0
		0-Bounded LE	0.101	228	2,673,846	723,760	81.93	13,313,800	0
		2-Bounded LE	0.091	219	2,673,847	723,758	81.93	11,160,163	0
		4-Bounded LE	0.089	211	2,673,847	723,758	81.93	9,790,146	0
		8-Bounded LE	0.089	211	2,673,847	723,758	81.93	9,790,146	0
		16-Bounded LE	0.089	211	2,673,847	723,758	81.93	9,790,146	0
		32-Bounded LE	0.091	206	2,673,847	723,758	81.93	9,568,892	0
		64-Bounded LE	0.091	206	2,673,847	723,758	81.93	9,568,892	0
		∞-Bounded LE	0.091	206	2,673,847	723,758	81.93	9,568,892	0
		0-Bounded	0.101	228	2,673,846	723,760	81.93	13,313,800	0
		2-Bounded	0.091	219	2,673,847	723,758	81.93	11,160,163	0
		4-Bounded	0.090	209	2,673,847	723,758	81.93	9,790,146	0
		8-Bounded	0.090	209	2,673,847	723,758	81.93	9,790,146	0
		16-Bounded	0.091	209	2,673,847	723,758	81.93	9,790,146	0
		32-Bounded	0.091	203	2,673,847	723,758	81.93	9,568,892	0
		64-Bounded	0.092	203	2,673,847	723,758	81.93	9,568,892	0
		∞-Bounded	0.092	203	2,673,847	723,758	81.93	9,568,892	0
		0CFA	0.084	144	2,673,848	723,691	81.94	6,870,658	0
		1-0CFA	0.083	144	2,673,848	723,691	81.94	6,870,658	0
		2-0CFA	0.083	144	2,673,848	723,691	81.94	6,870,658	0
		3-0CFA	0.084	144	2,673,848	723,691	81.94	6,870,658	0
		1-1CFA	0.086	135	2,673,848	723,691	81.94	6,870,658	0
		2-2CFA	0.083	144	2,673,848	723,691	81.94	6,870,658	0
		3-3CFA	0.084	144	2,673,848	723,691	81.94	6,870,658	0
		bounded-CPA	0.087	140	2,673,848	723,691	81.94	6,870,658	0
		SCS	0.087	140	2,673,848	723,691	81.94	6,870,658	0
		prof (optimisitc)	0.076	138	2,673,849	723,690	81.94	4,485,893	0

Table B.2: Application performance

	CGC Algorithm	Execution Time	Code Size	# of Calls	# of Sends	% Except Possible	# of Class Tests	# of Heap Alloc
deltablue (Cecil) — *without* profile-guided receiver class prediction	none (base)	1.990	444	5,880,597	48,350,790	100.00	80,712,421	0
	selector	1.953	306	5,880,597	48,350,790	99.99	80,712,421	0
	0-Bounded LE	1.441	292	6,420,675	22,602,463	99.98	98,068,476	0
	2-Bounded LE	0.811	258	6,420,676	10,754,031	99.96	56,077,428	0
	4-Bounded LE	0.749	244	6,420,676	10,737,531	99.96	49,007,299	0
	8-Bounded LE	0.726	244	6,420,676	10,737,531	99.96	49,007,299	0
	16-Bounded LE	0.724	244	6,420,676	10,737,531	99.96	49,007,299	0
	32-Bounded LE	0.595	233	6,420,676	6,339,775	99.95	36,595,548	0
	64-Bounded LE	0.578	233	6,420,676	6,339,775	99.95	36,595,548	0
	∞-Bounded LE	0.580	233	6,420,676	6,339,775	99.95	36,595,548	0
	0-Bounded	1.588	292	6,420,675	22,602,463	99.98	98,068,476	0
	2-Bounded	0.866	257	6,420,676	10,754,031	99.96	56,077,427	0
	4-Bounded	0.695	239	6,420,676	10,737,531	99.96	43,151,481	0
	8-Bounded	0.697	239	6,420,676	10,737,531	99.96	43,151,481	0
	16-Bounded	0.703	239	6,420,676	10,737,531	99.96	43,151,481	0
	32-Bounded	0.589	232	6,420,676	6,339,775	95.62	36,595,536	0
	64-Bounded	0.592	232	6,420,676	6,339,775	95.62	36,595,536	0
	∞-Bounded	0.593	232	6,420,676	6,339,775	95.62	36,595,536	0
	0CFA	0.403	168	6,690,728	27,500	99.89	33,747,691	0
	1-0CFA	0.401	168	6,690,728	27,500	99.89	33,747,691	0
	2-0CFA	0.423	166	6,690,728	27,500	99.89	33,747,691	0
	3-0CFA	0.394	166	6,690,728	27,500	99.89	33,747,684	0
	1-1CFA	0.411	157	6,690,728	27,500	99.89	33,747,684	0
	2-2CFA	0.399	165	6,690,728	27,500	99.89	33,747,684	0
	3-3CFA	0.398	165	6,690,728	27,500	99.89	33,747,684	0
	bounded-CPA	0.432	166	6,690,728	27,500	99.89	33,747,691	0
	SCS	0.427	166	6,690,728	27,500	99.89	33,747,691	0
	prof (optimisitc)	0.372	151	6,690,728	27,500	99.89	18,643,786	0
deltablue (Cecil) — *with* profile-guided receiver class prediction	none (base)	0.574	485	5,875,075	931,780	100.00	80,364,550	0
	selector	0.546	333	5,875,075	931,780	99.98	80,364,550	0
	0-Bounded LE	0.536	299	6,420,675	380,669	99.90	81,466,741	0
	2-Bounded LE	0.451	262	6,420,676	380,668	99.90	52,912,369	0
	4-Bounded LE	0.422	249	6,420,676	380,668	99.90	52,301,847	0
	8-Bounded LE	0.427	248	6,420,676	380,668	99.90	52,301,847	0
	16-Bounded LE	0.424	248	6,420,676	380,668	99.90	52,301,847	0
	32-Bounded LE	0.404	231	6,420,676	309,124	99.90	41,827,603	0
	64-Bounded LE	0.403	231	6,420,676	309,124	99.90	41,827,603	0
	∞-Bounded LE	0.404	231	6,420,676	309,124	99.90	41,827,603	0
	0-Bounded	0.536	299	6,420,675	380,669	99.90	81,466,741	0
	2-Bounded	0.443	262	6,420,676	380,668	99.90	52,912,368	0
	4-Bounded	0.412	244	6,420,676	380,668	99.90	46,446,029	0
	8-Bounded	0.418	243	6,420,676	380,668	99.90	46,446,029	0
	16-Bounded	0.417	243	6,420,676	380,668	99.90	46,446,029	0
	32-Bounded	0.388	229	6,420,676	309,124	99.90	41,805,041	0
	64-Bounded	0.394	229	6,420,676	309,124	99.90	41,805,041	0
	∞-Bounded	0.390	229	6,420,676	309,124	99.90	41,805,041	0
	0CFA	0.383	168	6,690,728	27,500	99.89	33,747,684	0
	1-0CFA	0.377	168	6,690,728	27,500	99.89	33,747,684	0
	2-0CFA	0.378	167	6,690,728	27,500	99.89	33,747,684	0
	3-0CFA	0.362	167	6,690,728	27,500	99.89	33,747,684	0
	1-1CFA	0.393	158	6,690,728	27,500	99.89	33,747,684	0
	2-2CFA	0.380	165	6,690,728	27,500	99.89	33,747,684	0
	3-3CFA	0.375	165	6,690,728	27,500	99.89	33,747,684	0
	bounded-CPA	0.373	167	6,690,728	27,500	99.89	33,747,684	0
	SCS	0.363	167	6,690,728	27,500	99.89	33,747,684	0
	prof (optimisitc)	0.335	150	6,690,728	27,500	99.89	18,643,786	0

Table B.2: Application performance

		CGC Algorithm	Execution Time	Code Size	# of Calls	# of Sends	% Except Possible	# of Class Tests	# of Heap Alloc
instr sched (Cecil)	*without* profile-guided receiver class prediction	none (base)	1.110	1,399	176,975	1,600,916	93.91	2,587,003	1,247
		selector	1.094	1,087	176,975	1,600,916	93.89	2,587,003	1,247
		0-Bounded LE	1.080	833	176,975	1,559,396	93.74	2,629,974	1,247
		2-Bounded LE	0.950	826	209,233	1,319,834	91.95	2,317,380	1,247
		4-Bounded LE	0.878	814	236,088	1,229,744	91.60	2,281,132	1,247
		8-Bounded LE	0.901	815	266,720	1,196,681	89.49	2,310,853	1,247
		16-Bounded LE	0.912	814	266,720	1,196,681	89.49	2,309,917	1,247
		32-Bounded LE	0.901	814	266,720	1,196,681	89.49	2,309,917	1,247
		64-Bounded LE	0.803	779	276,752	933,753	91.38	2,324,551	1,247
		∞-Bounded LE	0.799	779	276,752	933,753	91.38	2,324,551	1,247
		0-Bounded	1.061	833	176,975	1,559,396	93.74	2,629,974	1,247
		2-Bounded	0.936	822	209,233	1,319,834	91.95	2,317,374	1,247
		4-Bounded	0.878	813	236,088	1,229,744	91.60	2,281,122	1,247
		8-Bounded	0.906	815	266,720	1,196,681	89.49	2,309,644	1,247
		16-Bounded	0.909	814	266,720	1,196,681	89.49	2,308,708	1,247
		32-Bounded	0.792	778	276,752	933,753	91.38	2,319,017	1,247
		64-Bounded	0.828	776	276,752	933,753	91.38	2,319,017	1,247
		∞-Bounded	0.810	776	276,752	933,753	91.38	2,319,017	1,247
		0CFA	0.512	616	461,513	206,963	64.40	1,644,444	1,247
		1-0CFA	0.486	602	464,209	191,369	63.29	1,372,734	1,247
		2-0CFA	0.496	581	464,209	191,369	63.29	924,580	1,247
		3-0CFA	0.494	580	464,209	191,369	63.29	924,312	1,247
		1-1CFA	0.474	587	464,209	191,369	63.29	1,307,888	1,247
		2-2CFA	0.484	577	464,209	191,369	63.29	859,600	1,247
		3-3CFA	0.475	574	464,209	191,369	63.29	859,332	1,247
		bounded-CPA	0.502	575	464,209	191,369	63.29	924,241	1,247
		SCS	0.499	575	464,209	191,369	63.29	924,241	1,247
		prof (optimisitc)	0.425	477	487,516	90,450	61.54	790,707	1,247
	with profile-guided receiver class prediction	none (base)	0.628	1,423	254,121	362,694	99.16	2,631,409	1,247
		selector	0.613	1,105	254,121	362,694	99.11	2,631,409	1,247
		0-Bounded LE	0.611	852	254,121	362,109	99.10	2,632,932	1,247
		2-Bounded LE	0.557	831	277,353	311,398	96.15	2,182,602	1,247
		4-Bounded LE	0.549	820	285,375	302,151	96.15	2,175,517	1,247
		8-Bounded LE	0.563	819	285,375	302,146	96.15	2,175,512	1,247
		16-Bounded LE	0.551	819	285,375	302,146	96.15	2,174,576	1,247
		32-Bounded LE	0.555	819	285,375	302,146	96.15	2,174,576	1,247
		64-Bounded LE	0.563	773	295,407	287,170	96.58	2,059,507	1,247
		∞-Bounded LE	0.554	773	295,407	287,170	96.58	2,059,507	1,247
		0-Bounded	0.618	852	254,121	362,109	99.10	2,632,932	1,247
		2-Bounded	0.559	826	277,353	311,393	96.15	2,182,596	1,247
		4-Bounded	0.554	819	285,375	302,146	96.15	2,175,512	1,247
		8-Bounded	0.553	819	285,375	302,146	96.15	2,167,286	1,247
		16-Bounded	0.559	818	285,375	302,146	96.15	2,166,350	1,247
		32-Bounded	0.561	771	295,407	287,170	96.58	2,056,404	1,247
		64-Bounded	0.577	769	295,407	287,170	96.58	2,056,404	1,247
		∞-Bounded	0.580	769	295,407	287,170	96.58	2,056,404	1,247
		0CFA	0.470	609	457,774	118,666	68.34	1,686,041	1,247
		1-0CFA	0.492	598	460,470	114,868	67.81	1,405,836	1,247
		2-0CFA	0.468	576	460,470	113,760	67.74	958,022	1,247
		3-0CFA	0.477	576	460,470	113,760	67.74	957,754	1,247
		1-1CFA	0.479	583	460,734	114,671	67.76	1,341,581	1,247
		2-2CFA	0.466	573	460,734	113,496	67.70	893,834	1,247
		3-3CFA	0.463	570	460,734	113,496	67.70	893,566	1,247
		bounded-CPA	0.467	571	460,470	113,757	67.74	957,683	1,247
		SCS	0.477	571	460,470	113,757	67.74	957,683	1,247
		prof (optimisitc)	0.412	476	485,687	84,266	61.85	765,333	1,247

Table B.2: Application performance

		CGC Algorithm	Execution Time	Code Size	# of Calls	# of Sends	% Except Possible	# of Class Tests	# of Heap Alloc
typechecker (Cecil)	*without* profile-guided receiver class prediction	none (base)	9.739	4,168	2,171,507	9,045,278	99.96	15,032,605	231
		selector	9.762	3,662	2,171,507	9,045,278	94.44	15,032,031	231
		0-Bounded LE	10.066	3,450	2,172,464	8,981,947	94.89	15,088,717	231
		2-Bounded LE	8.465	3,329	2,294,344	6,747,276	93.55	11,930,767	231
		4-Bounded LE	7.879	3,300	2,402,417	6,478,242	93.43	11,865,916	231
		8-Bounded LE	7.923	3,291	2,402,417	6,478,182	93.43	11,829,933	231
		16-Bounded LE	8.117	3,291	2,402,417	6,478,182	93.43	11,829,933	231
		32-Bounded LE	7.802	3,291	2,402,417	6,478,182	93.43	11,829,933	231
		64-Bounded LE	8.114	3,291	2,402,417	6,478,182	93.43	11,829,933	231
		∞-Bounded LE	7.490	3,127	2,439,279	5,909,397	93.16	11,356,164	231
		0-Bounded	9.851	3,450	2,172,464	8,981,947	94.89	15,088,717	231
		2-Bounded	7.899	3,329	2,295,781	6,697,412	93.50	11,885,849	231
		4-Bounded	7.877	3,296	2,402,417	6,477,730	93.42	11,865,519	231
		8-Bounded	7.839	3,289	2,402,417	6,477,670	93.41	11,829,536	231
		16-Bounded	7.733	3,289	2,402,417	6,477,670	93.41	11,829,536	231
		32-Bounded	7.836	3,289	2,402,417	6,477,670	93.41	11,829,536	231
		64-Bounded	7.690	3,289	2,402,417	6,477,670	93.41	11,829,536	231
		∞-Bounded	7.393	3,064	2,450,228	5,661,825	90.16	11,380,385	231
		0CFA	6.893	3,203	2,475,313	4,967,580	89.71	10,604,554	231
		1-0CFA	5.876	2,860	2,762,950	3,787,570	75.36	9,263,567	231
		2-0CFA							
		3-0CFA							
		1-1CFA							
		2-2CFA							
		3-3CFA							
		bounded-CPA							
		SCS							
		prof (optimisitc)	4.494	2,082	2,970,834	1,634,540	84.75	3,102,526	231
	with profile-guided receiver class prediction	none (base)	6.427	4,429	1,969,404	2,607,936	99.98	15,387,089	231
		selector	6.391	3,862	1,969,404	2,607,936	94.69	15,386,515	231
		0-Bounded LE	6.427	3,626	1,969,670	2,605,944	94.62	15,378,981	231
		2-Bounded LE	5.875	3,457	2,462,136	2,084,497	93.79	12,768,166	231
		4-Bounded LE	5.961	3,413	2,428,493	2,146,216	93.85	12,702,580	231
		8-Bounded LE	5.844	3,406	2,428,509	2,145,981	93.85	12,668,710	231
		16-Bounded LE	5.975	3,406	2,428,509	2,145,981	93.85	12,668,710	231
		32-Bounded LE	6.032	3,406	2,428,509	2,145,981	93.85	12,668,710	231
		64-Bounded LE	6.013	3,406	2,428,509	2,145,981	93.85	12,668,710	231
		∞-Bounded LE	5.729	3,205	2,438,475	2,195,940	93.93	12,316,024	231
		0-Bounded	6.254	3,626	1,969,670	2,605,944	94.62	15,378,981	231
		2-Bounded	5.853	3,456	2,463,557	2,082,841	93.77	12,769,486	231
		4-Bounded	5.806	3,409	2,428,493	2,146,216	93.83	12,700,685	231
		8-Bounded	5.839	3,401	2,428,509	2,145,981	93.83	12,666,836	231
		16-Bounded	5.838	3,401	2,428,509	2,145,981	93.83	12,666,836	231
		32-Bounded	5.743	3,401	2,428,509	2,145,981	93.83	12,666,836	231
		64-Bounded	5.861	3,401	2,428,509	2,145,981	93.83	12,666,836	231
		∞-Bounded	5.640	3,151	2,462,778	2,161,835	93.81	12.195,284	231
		0CFA	5.550	2,933	2,461,005	2,109,667	91.39	11,441,515	231
		1-0CFA	5.325	2,853	2,586,352	1,881,872	91.24	9,997,527	231
		2-0CFA							
		3-0CFA							
		1-1CFA							
		2-2CFA							
		3-3CFA							
		bounded-CPA							
		SCS							
		prof (optimisitc)	4.434	2,099	2,777,632	1,626,046	85.17	3,241,194	231

Table B.2: Application performance

	CGC Algorithm	Execution Time	Code Size	# of Calls	# of Sends	% Except Possible	# of Class Tests	# of Heap Alloc
new-tc (Cecil), *without* profile-guided receiver class prediction	none (base)	9.703	4,406	2,422,372	10,219,578	99.79	15,593,530	231
	selector	9.578	3,890	2,422,372	10,219,578	93.78	15,592,956	231
	0-Bounded LE	9.462	3,702	2,423,329	10,208,792	93.82	15,594,355	231
	2-Bounded LE	8.147	3,571	2,556,603	7,932,949	92.20	12,491,807	231
	4-Bounded LE	8.142	3,539	2,571,344	7,872,301	92.17	12,346,379	231
	8-Bounded LE	8.079	3,535	2,571,458	7,849,911	92.15	12,356,133	231
	16-Bounded LE	8.185	3,535	2,571,458	7,849,911	92.15	12,356,133	231
	32-Bounded LE	8.053	3,535	2,571,458	7,849,911	92.15	12,356,133	231
	64-Bounded LE	8.053	3,535	2,571,458	7,849,911	92.15	12,356,133	231
	∞-Bounded LE	7.900	3,365	2,606,841	7,210,770	90.69	12,056,840	231
	0-Bounded	9.377	3,703	2,423,329	10,208,792	93.82	15,594,355	231
	2-Bounded	8.276	3,571	2,559,458	7,927,487	92.19	12,492,567	231
	4-Bounded	8.083	3,535	2,571,344	7,871,789	92.16	12,344,741	231
	8-Bounded	8.017	3,532	2,571,458	7,849,399	92.14	12,355,737	231
	16-Bounded	8.182	3,532	2,571,458	7,849,399	92.14	12,355,737	231
	32-Bounded	8.205	3,532	2,571,458	7,849,399	92.14	12,355,737	231
	64-Bounded	8.116	3,532	2,571,458	7,849,399	92.14	12,355,737	231
	∞-Bounded	7.730	3,301	2,682,726	6,792,572	89.91	12,096,632	231
	0CFA	6.834	3,396	2,795,036	5,757,461	89.32	10,926,310	231
	1-0CFA	6.400	3,056	2,828,396	4,800,992	76.19	9,515,361	231
	2-0CFA							
	3-0CFA							
	1-1CFA							
	2-2CFA							
	3-3CFA							
	bounded-CPA							
	SCS							
	prof (optimisitc)	5.059	2,226	2,915,486	1,916,612	84.78	3,712,575	231
new-tc (Cecil), *with* profile-guided receiver class prediction	none (base)	7.288	4,537	1,997,231	3,050,698	99.88	16,685,424	231
	selector	7.045	4,003	1,997,231	3,050,698	95.60	16,684,850	231
	0-Bounded LE	6.875	3,804	1,997,497	3,045,950	95.67	16,679,778	231
	2-Bounded LE	6.515	3,689	2,486,987	2,513,560	94.90	14,332,677	231
	4-Bounded LE	6.423	3,632	2,496,716	2,487,171	94.90	14,199,002	231
	8-Bounded LE	6.243	3,629	2,496,830	2,486,118	94.90	14,197,320	231
	16-Bounded LE	6.194	3,629	2,496,830	2,486,118	94.90	14,197,320	231
	32-Bounded LE	6.291	3,629	2,496,830	2,486,118	94.90	14,197,320	231
	64-Bounded LE	6.276	3,629	2,496,830	2,486,118	94.90	14,197,320	231
	∞-Bounded LE	6.243	3,439	2,529,568	2,467,400	94.96	14,140,865	231
	0-Bounded	6.816	3,804	1,997,497	3,045,950	95.66	16,679,778	231
	2-Bounded	6.473	3,686	2,489,842	2,510,524	94.88	14,335,328	231
	4-Bounded	6.367	3,627	2,496,716	2,487,171	94.88	14,195,896	231
	8-Bounded	6.356	3,624	2,496,830	2,486,118	94.88	14,195,470	231
	16-Bounded	6.326	3,624	2,496,830	2,486,118	94.88	14,195,470	231
	32-Bounded	6.262	3,624	2,496,830	2,486,118	94.88	14,195,470	231
	64-Bounded	6.321	3,624	2,496,830	2,486,118	94.88	14,195,470	231
	∞-Bounded	6.226	3,360	2,599,551	2,368,802	94.79	13,943,693	231
	0CFA	5.969	3,156	2,707,116	2,170,160	93.44	13,099,499	231
	1-0CFA	5.800	3,050	2,707,208	2,160,824	92.87	10,882,929	231
	2-0CFA							
	3-0CFA							
	1-1CFA							
	2-2CFA							
	3-3CFA							
	bounded-CPA							
	SCS							
	prof (optimisitc)	5.243	2,246	2,874,050	1,906,240	85.41	4,012,514	231

Table B.2: Application performance

	CGC Algorithm	Execution Time	Code Size	# of Calls	# of Sends	% Except Possible	# of Class Tests	# of Heap Alloc
compiler (Cecil), *without* profile-guided receiver class prediction	none (base)	200.440	7,851	22,389,339	186,546,159	97.04	211,311,525	315,788
	selector	197.467	7,466	22,389,339	186,544,453	94.45	211,300,797	315,788
	0-Bounded LE	196.745	7,295	22,407,664	186,266,063	94.46	211,451,453	315,788
	2-Bounded LE	177.459	7,040	24,719,574	171,988,427	94.19	192,174,410	315,788
	4-Bounded LE	172.722	6,941	24,907,714	165,985,634	94.10	184,607,722	315,788
	8-Bounded LE	173.014	6,927	24,911,130	165,899,194	94.10	184,557,931	315,788
	16-Bounded LE	175.935	6,927	24,911,130	165,899,194	94.10	184,557,931	315,788
	32-Bounded LE	174.980	6,927	24,911,130	165,899,194	94.10	184,557,931	315,788
	64-Bounded LE	174.693	6,927	24,911,130	165,899,194	94.10	184,557,931	315,788
	∞-Bounded LE	169.041	6,737	29,657,749	151,270,628	94.00	176,645,825	315,788
	0-Bounded	194.137	7,311	22,407,837	186,266,194	94.46	211,451,672	315,788
	2-Bounded	179.647	7,035	24,726,022	169,488,251	94.11	189,658,964	315,788
	4-Bounded	172.633	6,929	24,911,671	165,836,104	94.10	184,249,329	315,788
	8-Bounded	175.168	6,917	24,911,671	165,786,962	94.10	184,176,509	315,788
	16-Bounded	174.773	6,917	24,911,671	165,786,962	94.10	184,176,509	315,788
	32-Bounded	174.630	6,917	24,911,671	165,786,962	94.10	184,176,509	315,788
	64-Bounded	174.619	6,917	24,911,671	165,786,962	94.10	184,176,509	315,788
	∞-Bounded							
	0CFA	156.376	6,716	29,751,078	141,944,194	92.32	167,370,996	315,788
	1-0CFA							
	2-0CFA							
	3-0CFA							
	1-1CFA							
	2-2CFA							
	3-3CFA							
	bounded-CPA							
	SCS							
	prof (optimisitc)	95.835	4,177	35,716,641	35,628,092	85.74	54,540,410	314,961
compiler (Cecil), *with* profile-guided receiver class prediction	none (base)	124.527	7,854	21,432,200	44,628,848	97.81	211,025,488	315,788
	selector	123.762	7,459	21,432,200	44,627,142	94.34	211,020,104	315,788
	0-Bounded LE	120.642	7,287	21,674,715	44,379,349	94.34	211,051,991	315,788
	2-Bounded LE	114.591	7,069	24,334,928	41,343,555	94.14	190,585,029	315,788
	4-Bounded LE	115.998	6,941	24,376,989	41,163,145	94.18	188,212,236	315,788
	8-Bounded LE	114.165	6,939	24,382,126	41,136,718	94.20	188,128,319	315,788
	16-Bounded LE	113.842	6,939	24,382,126	41,136,718	94.20	188,128,319	315,788
	32-Bounded LE	113.448	6,939	24,382,126	41,136,718	94.20	188,128,319	315,788
	64-Bounded LE	114.885	6,939	24,382,126	41,136,718	94.20	188,128,319	315,788
	∞-Bounded LE	113.929	6,787	28,976,949	38,853,302	94.47	187,048,427	315,788
	0-Bounded	119.847	7,277	21,674,715	44,379,349	94.34	211,051,991	315,788
	2-Bounded	113.311	7,069	24,328,802	41,334,439	94.10	190,376,680	315,788
	4-Bounded	114.065	6,929	24,380,821	41,144,653	94.14	188,004,236	315,788
	8-Bounded	114.481	6,928	24,382,667	41,126,884	94.16	187,958,225	315,788
	16-Bounded	114.264	6,928	24,382,667	41,126,884	94.16	187,958,225	315,788
	32-Bounded	114.002	6,928	24,382,667	41,126,884	94.16	187,958,225	315,788
	64-Bounded	114.255	6,928	24,382,667	41,126,884	94.16	187,958,225	315,788
	∞-Bounded							
	0CFA	111.385	6,534	29,214,181	38,176,223	93.51	202,564,279	315,788
	1-0CFA							
	2-0CFA							
	3-0CFA							
	1-1CFA							
	2-2CFA							
	3-3CFA							
	bounded-CPA							
	SCS							
	prof (optimisitc)	92.220	4,191	34,632,808	29,028,413	88.80	78,991,338	314,961

Table B.2: Application performance

		CGC Algorithm	Execution Time	Code Size	# of Calls	# of Sends	% Except Possible	# of Class Tests	# of Heap Alloc
cassowary (Java)	*without* profile-guided receiver class prediction	none (base)	8.156	988	7,946,309	6,844,940	100.00	4,506,698	0
		selector	8.387	742	7,946,309	6,844,940	94.41	4,506,698	0
		0-Bounded LE	8.578	695	7,946,347	6,844,864	94.41	4,506,850	0
		2-Bounded LE	8.599	694	7,946,346	6,844,863	94.41	4,506,850	0
		4-Bounded LE	8.188	692	7,947,010	6,844,161	94.41	4,507,476	0
		8-Bounded LE	8.183	692	7,947,010	6,844,161	94.41	4,507,476	0
		16-Bounded LE	7.594	691	7,947,025	3,776,549	92.95	7,573,996	0
		32-Bounded LE	6.773	686	7,947,030	637,333	90.37	13,852,146	0
		64-Bounded LE	6.851	686	7,947,032	637,333	90.37	13,852,147	0
		∞-Bounded LE	7.206	686	7,947,034	637,333	90.37	13,852,148	0
		0-Bounded	8.860	695	7,946,347	6,844,864	94.41	4,506,850	0
		2-Bounded	8.743	694	7,946,345	6,844,862	94.41	4,506,850	0
		4-Bounded	8.617	692	7,947,008	6,844,161	94.41	4,507,475	0
		8-Bounded	8.690	692	7,947,012	6,844,161	94.41	4,507,477	0
		16-Bounded	7.060	686	7,947,030	637,333	90.37	13,852,146	0
		32-Bounded	7.118	686	7,947,028	637,333	90.37	13,852,145	0
		64-Bounded	6.641	686	7,947,034	637,333	90.37	13,852,148	0
		∞-Bounded	7.039	686	7,947,030	637,333	90.37	13,852,146	0
		0CFA	6.566	694	7,947,032	637,333	90.37	13,847,582	0
		1-0CFA	6.702	680	7,947,032	637,333	90.37	13,847,582	0
		2-0CFA	6.701	680	7,947,032	637,333	90.37	13,847,582	0
		3-0CFA	6.725	680	7,947,030	637,333	90.37	13,847,581	0
		1-1CFA	6.894	668	7,947,032	637,333	90.37	10,707,594	0
		2-2CFA	6.696	668	7,947,032	637,333	90.37	10,707,594	0
		3-3CFA	6.656	668	7,947,030	637,333	90.37	10,707,594	0
		bounded-CPA	6.946	680	7,947,032	637,333	90.37	13,847,582	0
		SCS	6.678	681	7,947,036	637,333	90.37	13,847,584	0
		prof (optimisitc)	6.365	653	7,947,032	1,791	97.59	10,129,031	0
	with profile-guided receiver class prediction	none (base)	6.620	989	7,946,308	33,547	100.00	15,959,427	0
		selector	6.751	744	7,946,305	33,544	97.58	15,959,427	0
		0-Bounded LE	6.911	696	7,946,347	33,472	97.58	15,959,541	0
		2-Bounded LE	6.902	695	7,946,347	33,472	97.58	15,959,541	0
		4-Bounded LE	6.821	693	7,947,008	32,769	97.58	15,960,204	0
		8-Bounded LE	6.922	693	7,947,012	32,769	97.58	15,960,206	0
		16-Bounded LE	6.549	691	7,947,025	14,008	97.57	13,609,128	0
		32-Bounded LE	6.847	687	7,947,032	3,757	97.57	14,621,335	0
		64-Bounded LE	6.828	687	7,947,032	3,757	97.57	14,621,335	0
		∞-Bounded LE	6.917	687	7,947,032	3,757	97.57	14,621,335	0
		0-Bounded	6.876	696	7,946,345	33,470	97.58	15,959,541	0
		2-Bounded	6.886	695	7,946,346	33,471	97.58	15,959,541	0
		4-Bounded	6.840	693	7,947,006	32,769	97.58	15,960,203	0
		8-Bounded	6.916	693	7,947,008	32,769	97.58	15,960,204	0
		16-Bounded	6.648	687	7,947,034	3,757	97.57	14,621,336	0
		32-Bounded	6.633	686	7,947,032	3,757	97.57	14,621,335	0
		64-Bounded	6.651	686	7,947,032	3,757	97.57	14,621,335	0
		∞-Bounded	6.719	686	7,947,034	3,757	97.57	14,621,336	0
		0CFA	6.612	681	7,947,032	3,757	97.57	14,616,770	0
		1-0CFA	6.800	681	7,947,032	3,757	97.57	14,616,770	0
		2-0CFA	6.856	681	7,947,030	3,757	97.57	14,616,769	0
		3-0CFA	6.763	681	7,947,032	3,757	97.57	14,616,770	0
		1-1CFA	6.586	668	7,947,032	3,757	97.57	11,476,782	0
		2-2CFA	6.589	668	7,947,036	3,757	97.57	11,476,782	0
		3-3CFA	6.594	668	7,947,034	3,757	97.57	11,476,782	0
		bounded-CPA	6.832	681	7,947,034	3,757	97.57	14,616,771	0
		SCS	6.616	681	7,947,032	3,757	97.57	14,616,770	0
		prof (optimisitc)	6.616	653	7,947,032	1,791	97.59	9,762,903	0

Table B.2: Application performance

		CGC Algorithm	Execution Time	Code Size	# of Calls	# of Sends	% Except Possible	# of Class Tests	# of Heap Alloc
toba (Java)	*without* profile-guided receiver class prediction	none (base)	1.340	1,124	1,198,993	489,250	100.00	271,968	0
		selector	1.316	916	1,197,505	482,678	92.59	271,658	0
		0-Bounded LE	1.275	886	1,197,667	482,205	92.59	247,196	0
		2-Bounded LE	1.266	886	1,199,217	488,653	92.60	247,692	0
		4-Bounded LE	1.239	880	1,581,678	106,012	92.60	630,060	0
		8-Bounded LE	1.262	880	1,581,678	106,012	92.60	630,060	0
		16-Bounded LE	1.247	880	1,581,678	106,012	92.60	630,060	0
		32-Bounded LE	1.254	880	1,581,678	106,012	92.60	630,060	0
		64-Bounded LE	1.244	878	1,585,843	101,847	92.60	633,158	0
		∞-Bounded LE	1.280	878	1,585,843	101,847	92.60	633,158	0
		0-Bounded	1.316	886	1,199,217	488,653	92.60	247,692	0
		2-Bounded	1.283	886	1,199,217	488,653	92.60	247,692	0
		4-Bounded	1.259	880	1,581,678	106,012	92.60	630,060	0
		8-Bounded	1.260	880	1,581,678	106,012	92.60	630,060	0
		16-Bounded	1.263	880	1,581,678	106,012	92.60	630,060	0
		32-Bounded	1.265	880	1,581,678	106,012	92.60	630,060	0
		64-Bounded	1.279	878	1,585,843	101,847	92.60	633,158	0
		∞-Bounded	1.304	878	1,585,843	101,847	92.60	633,158	0
		0CFA	1.307	881	1,581,195	98,497	92.59	629,564	0
		1-0CFA	1.264	873	1,585,843	101,847	92.60	633,158	0
		2-0CFA	1.269	873	1,585,843	101,847	92.60	633,158	0
		3-0CFA	1.243	873	1,585,843	101,847	92.60	633,158	0
		1-1CFA	1.219	874	1,641,564	43,197	92.59	299,450	0
		2-2CFA	1.232	873	1,641,564	43,197	92.59	266,102	0
		3-3CFA	1.223	873	1,641,564	43,197	92.59	266,102	0
		bounded-CPA	1.220	879	1,585,843	101,847	92.60	633,158	0
		SCS	1.241	879	1,584,293	95,399	92.59	632,662	0
		prof (optimisitc)	1.153	813	1,633,628	43,190	92.56	265,390	0
	with profile-guided receiver class prediction	none (base)	1.316	1,126	1,198,993	488,182	100.00	273,238	0
		selector	1.324	917	1,198,993	488,182	92.60	273,235	0
		0-Bounded LE	1.335	888	1,197,667	481,137	92.59	248,332	0
		2-Bounded LE	1.314	887	1,199,217	487,585	92.60	248,828	0
		4-Bounded LE	1.236	882	1,581,678	104,944	92.60	631,258	0
		8-Bounded LE	1.246	881	1,581,678	104,944	92.60	631,258	0
		16-Bounded LE	1.273	881	1,581,678	104,944	92.60	631,258	0
		32-Bounded LE	1.284	881	1,581,678	104,944	92.60	631,258	0
		64-Bounded LE	1.227	879	1,585,843	100,779	92.60	634,356	0
		∞-Bounded LE	1.249	879	1,585,843	100,779	92.60	634,356	0
		0-Bounded	1.295	888	1,199,217	487,585	92.60	248,828	0
		2-Bounded	1.288	887	1,199,217	487,585	92.60	248,828	0
		4-Bounded	1.253	882	1,581,678	104,944	92.60	631,258	0
		8-Bounded	1.253	881	1,581,678	104,944	92.60	631,258	0
		16-Bounded	1.261	881	1,581,678	104,944	92.60	631,258	0
		32-Bounded	1.236	881	1,581,678	104,944	92.60	631,258	0
		64-Bounded	1.250	879	1,585,843	100,779	92.60	634,356	0
		∞-Bounded	1.260	879	1,585,843	100,779	92.60	634,356	0
		0CFA	1.222	880	1,585,843	100,779	92.60	634,356	0
		1-0CFA	1.216	874	1,585,843	100,779	92.60	634,356	0
		2-0CFA	1.186	874	1,585,843	100,779	92.60	634,356	0
		3-0CFA	1.231	874	1,585,843	100,779	92.60	634,356	0
		1-1CFA	1.213	874	1,641,564	43,197	92.59	292,933	0
		2-2CFA	1.217	873	1,641,564	43,197	92.59	266,102	0
		3-3CFA	1.189	873	1,633,628	43,197	92.58	259,344	0
		bounded-CPA	1.191	879	1,585,843	100,779	92.60	634,356	0
		SCS	1.231	880	1,585,843	100,779	92.60	634,356	0
		prof (optimisitc)	1.160	813	1,633,628	43,190	92.56	256,594	0

Table B.2: Application performance

	CGC Algorithm	Execution Time	Code Size	# of Calls	# of Sends	% Except Possible	# of Class Tests	# of Heap Alloc
java-cup (Java) *without* profile-guided receiver class prediction	none (base)	0.216	1,262	207,260	58,169	100.00	81,608	0
	selector	0.218	1,031	207,260	58,169	88.77	81,608	0
	0-Bounded LE	0.218	1,002	223,162	42,116	88.77	97,695	0
	2-Bounded LE	0.214	1,001	223,162	42,116	88.77	97,695	0
	4-Bounded LE	0.206	998	260,078	5,178	88.76	118,710	0
	8-Bounded LE	0.208	998	260,078	5,178	88.76	118,710	0
	16-Bounded LE	0.209	998	260,078	5,178	88.76	118,710	0
	32-Bounded LE	0.212	985	260,078	4,868	88.75	118,693	0
	64-Bounded LE	0.210	985	260,078	4,868	88.75	118,693	0
	∞-Bounded LE	0.208	985	260,078	4,868	88.75	118,693	0
	0-Bounded	0.233	1,002	223,162	42,116	88.77	97,695	0
	2-Bounded	0.216	1,001	223,162	42,116	88.77	97,695	0
	4-Bounded	0.207	998	260,078	5,178	88.76	118,710	0
	8-Bounded	0.209	998	260,078	5,178	88.76	118,710	0
	16-Bounded	0.209	998	260,078	5,178	88.76	118,710	0
	32-Bounded	0.206	985	260,078	4,868	88.75	117,771	0
	64-Bounded	0.208	985	260,078	4,868	88.75	117,771	0
	∞-Bounded	0.206	985	260,078	4,868	88.75	117,771	0
	0CFA	0.205	990	260,078	4,868	88.75	116,100	0
	1-0CFA	0.207	973	260,078	4,868	88.75	114,697	0
	2-0CFA	0.200	973	260,078	4,868	88.75	114,697	0
	3-0CFA	0.203	973	260,078	4,868	88.75	114,697	0
	1-1CFA	0.207	972	260,078	4,868	88.75	77,727	0
	2-2CFA	0.203	972	260,078	4,868	88.75	77,600	0
	3-3CFA	0.204	972	260,078	4,868	88.75	77,600	0
	bounded-CPA	0.206	973	260,078	4,868	88.75	114,697	0
	SCS	0.203	973	260,078	4,868	88.75	114,697	0
	prof (optimisitc)	0.201	850	260,103	4,532	88.68	76,699	0
java-cup (Java) *with* profile-guided receiver class prediction	none (base)	0.213	1,263	207,260	56,386	100.00	88,105	0
	selector	0.217	1,031	207,260	56,386	88.70	88,105	0
	0-Bounded LE	0.225	1,002	223,162	40,333	88.69	104,192	0
	2-Bounded LE	0.214	1,001	223,162	40,333	88.69	104,192	0
	4-Bounded LE	0.206	999	260,078	3,395	88.69	125,207	0
	8-Bounded LE	0.206	999	260,078	3,395	88.69	125,207	0
	16-Bounded LE	0.206	999	260,078	3,395	88.69	125,207	0
	32-Bounded LE	0.203	985	260,078	3,085	88.68	125,190	0
	64-Bounded LE	0.202	985	260,078	3,085	88.68	125,190	0
	∞-Bounded LE	0.200	985	260,078	3,085	88.68	125,190	0
	0-Bounded	0.211	1,002	223,162	40,333	88.69	104,192	0
	2-Bounded	0.213	1,001	223,162	40,333	88.69	104,192	0
	4-Bounded	0.206	999	260,078	3,395	88.69	125,207	0
	8-Bounded	0.204	999	260,078	3,395	88.69	125,207	0
	16-Bounded	0.204	999	260,078	3,395	88.69	125,207	0
	32-Bounded	0.210	985	260,078	3,085	88.68	124,268	0
	64-Bounded	0.207	985	260,078	3,085	88.68	124,268	0
	∞-Bounded	0.208	985	260,078	3,085	88.68	124,268	0
	0CFA	0.205	974	260,078	3,085	88.68	122,580	0
	1-0CFA	0.203	976	260,080	3,085	88.67	121,194	0
	2-0CFA	0.203	976	260,080	3,085	88.67	121,194	0
	3-0CFA	0.204	976	260,080	3,085	88.67	121,194	0
	1-1CFA	0.207	975	260,080	3,085	88.67	84,224	0
	2-2CFA	0.201	975	260,080	3,085	88.67	84,097	0
	3-3CFA	0.202	975	260,080	3,085	88.67	84,097	0
	bounded-CPA	0.203	976	260,080	3,085	88.67	121,194	0
	SCS	0.208	976	260,080	3,085	88.67	121,194	0
	prof (optimisitc)	0.200	850	260,103	2,626	88.60	83,729	0

Table B.2: Application performance

	CGC Algorithm	Execution Time	Code Size	# of Calls	# of Sends	% Except Possible	# of Class Tests	# of Heap Alloc
espresso (Java), *without* profile-guided receiver class prediction	none (base)	1.678	1,721	794,467	209,745	100.00	438,018	0
	selector	1.666	1,529	794,476	209,745	89.12	437,691	0
	0-Bounded LE	1.667	1,503	794,476	209,745	89.12	437,691	0
	2-Bounded LE	1.661	1,500	794,467	209,745	89.12	437,691	0
	4-Bounded LE	1.551	1,488	794,519	209,534	89.12	437,542	0
	8-Bounded LE	1.551	1,486	794,519	209,534	89.12	437,542	0
	16-Bounded LE	1.613	1,486	794,519	209,534	89.12	437,542	0
	32-Bounded LE	1.598	1,486	794,519	209,534	89.12	437,542	0
	64-Bounded LE	1.592	1,486	794,519	209,534	89.12	437,542	0
	∞-Bounded LE	1.610	1,478	794,519	209,534	89.12	435,853	0
	0-Bounded	1.670	1,503	794,467	209,745	89.12	437,691	0
	2-Bounded	1.590	1,496	794,467	209,745	89.12	437,691	0
	4-Bounded	1.603	1,486	794,519	209,534	89.12	437,542	0
	8-Bounded	1.603	1,486	794,519	209,534	89.12	437,542	0
	16-Bounded	1.550	1,486	794,519	209,534	89.12	437,542	0
	32-Bounded	1.551	1,486	794,519	209,534	89.12	437,542	0
	64-Bounded	1.549	1,486	794,519	209,534	89.12	437,542	0
	∞-Bounded	1.555	1,478	794,519	209,534	89.12	435,853	0
	0CFA	1.677	1,478	794,528	209,534	89.12	435,939	0
	1-0CFA	1.654	1,469	794,519	209,534	89.12	435,853	0
	2-0CFA	1.657	1,469	794,519	209,534	89.12	435,853	0
	3-0CFA							
	1-1CFA	1.590	1,473	794,571	209,482	89.12	428,484	0
	2-2CFA							
	3-3CFA							
	bounded-CPA	1.583	1,475	794,519	209,534	89.12	435,853	0
	SCS	1.602	1,475	794,528	209,534	89.12	435,853	0
	prof (optimisitc)	1.521	1,331	814,130	170,778	89.07	423,822	0
espresso (Java), *with* profile-guided receiver class prediction	none (base)	1.713	1,726	795,698	184,324	100.00	491,283	0
	selector	1.657	1,534	795,698	184,324	89.09	490,956	0
	0-Bounded LE	1.683	1,507	795,707	184,324	89.09	490,956	0
	2-Bounded LE	1.661	1,504	795,698	184,324	89.09	490,956	0
	4-Bounded LE	1.576	1,492	795,750	184,113	89.09	490,963	0
	8-Bounded LE	1.599	1,490	795,750	184,113	89.09	490,963	0
	16-Bounded LE	1.597	1,490	795,750	184,113	89.09	490,963	0
	32-Bounded LE	1.597	1,490	795,750	184,113	89.09	490,963	0
	64-Bounded LE	1.597	1,490	795,750	184,113	89.09	490,963	0
	∞-Bounded LE	1.677	1,482	795,759	184,113	89.09	489,274	0
	0-Bounded	1.685	1,507	795,698	184,324	89.09	490,956	0
	2-Bounded	1.656	1,501	795,698	184,324	89.09	490,956	0
	4-Bounded	1.659	1,490	795,750	184,113	89.09	490,963	0
	8-Bounded	1.661	1,490	795,750	184,113	89.09	490,963	0
	16-Bounded	1.551	1,490	795,750	184,113	89.09	490,963	0
	32-Bounded	1.552	1,490	795,750	184,113	89.09	490,963	0
	64-Bounded	1.551	1,490	795,750	184,113	89.09	490,963	0
	∞-Bounded	1.582	1,482	795,759	184,113	89.09	489,274	0
	0CFA	1.532	1,479	795,750	184,113	89.09	489,274	0
	1-0CFA	1.592	1,473	795,759	184,113	89.09	489,274	0
	2-0CFA	1.638	1,473	795,750	184,113	89.09	489,274	0
	3-0CFA							
	1-1CFA	1.584	1,477	795,802	184,061	89.09	481,905	0
	2-2CFA							
	3-3CFA							
	bounded-CPA	1.533	1,479	795,750	184,113	89.09	489,274	0
	SCS	1.635	1,479	795,759	184,113	89.09	489,274	0
	prof (optimisitc)	1.526	1,334	815,352	152,231	88.87	468,733	0

Table B.2: Application performance

		CGC Algorithm	Execution Time	Code Size	# of Calls	# of Sends	% Except Possible	# of Class Tests	# of Heap Alloc
javac (Java)	*without* profile-guided receiver class prediction	none (base)	1.075	2,347	601,694	340,851	100.00	222,830	0
		selector	1.060	2,127	1,002,966	670,470	95.88	382,202	0
		0-Bounded LE	1.022	2,095	601,708	340,766	94.45	222,209	0
		2-Bounded LE	1.028	2,094	601,694	340,766	94.45	222,209	0
		4-Bounded LE	1.017	2,088	601,694	340,766	94.45	221,465	0
		8-Bounded LE	1.095	2,088	601,708	340,766	94.45	221,465	0
		16-Bounded LE	1.097	2,088	601,694	340,766	94.45	221,465	0
		32-Bounded LE	1.096	2,088	601,694	340,766	94.45	221,465	0
		64-Bounded LE	1.097	2,088	601,694	340,766	94.45	221,465	0
		∞-Bounded LE	1.038	2,071	601,694	340,766	94.45	220,811	0
		0-Bounded	1.022	2,095	601,694	340,766	94.45	222,209	0
		2-Bounded	1.025	2,094	601,694	340,766	94.45	222,209	0
		4-Bounded	1.017	2,088	601,694	340,766	94.45	221,465	0
		8-Bounded	1.083	2,088	601,694	340,766	94.45	221,465	0
		16-Bounded	1.021	2,088	601,694	340,766	94.45	221,465	0
		32-Bounded	1.021	2,088	601,694	340,766	94.45	221,465	0
		64-Bounded	1.086	2,088	601,694	340,766	94.45	221,465	0
		∞-Bounded	1.044	2,071	601,694	340,766	94.45	220,811	0
		0CFA	1.051	2,069	601,708	340,766	94.45	220,811	0
		1-0CFA	1.021	2,058	601,708	340,766	94.45	220,811	0
		2-0CFA							
		3-0CFA							
		1-1CFA	0.991	2,063	796,389	145,710	94.45	256,581	0
		2-2CFA							
		3-3CFA							
		bounded-CPA	1.031	2,068	601,694	340,766	94.45	220,811	0
		SCS	1.047	2,068	601,708	340,766	94.45	220,811	0
		prof (optimisitc)	0.949	1,725	842,864	92,720	94.81	164,514	0
	with profile-guided receiver class prediction	none (base)	1.030	2,349	602,056	311,734	100.00	305,188	0
		selector	1.010	2,128	1,003,386	631,825	96.50	496,575	0
		0-Bounded LE	1.007	2,097	602,070	311,649	95.41	304,567	0
		2-Bounded LE	1.022	2,095	602,056	311,649	95.41	304,567	0
		4-Bounded LE	0.994	2,089	602,056	311,649	95.41	303,823	0
		8-Bounded LE	1.008	2,089	602,056	311,649	95.41	303,823	0
		16-Bounded LE	1.007	2,089	602,056	311,649	95.41	303,823	0
		32-Bounded LE	1.002	2,089	602,056	311,649	95.41	303,823	0
		64-Bounded LE	1.002	2,089	602,056	311,649	95.41	303,823	0
		∞-Bounded LE	1.010	2,072	602,056	311,649	95.41	303,496	0
		0-Bounded	1.006	2,097	602,056	311,649	95.41	304,567	0
		2-Bounded	1.005	2,095	602,056	311,649	95.41	304,567	0
		4-Bounded	1.039	2,089	602,056	311,649	95.41	303,823	0
		8-Bounded	1.001	2,089	602,056	311,649	95.41	303,823	0
		16-Bounded	1.005	2,089	602,056	311,649	95.41	303,823	0
		32-Bounded	0.993	2,089	602,056	311,649	95.41	303,823	0
		64-Bounded	1.001	2,089	602,056	311,649	95.41	303,823	0
		∞-Bounded	1.027	2,072	602,056	311,649	95.41	303,496	0
		0CFA	1.040	2,070	602,056	311,649	95.41	303,496	0
		1-0CFA	1.001	2,059	602,056	311,649	95.41	303,496	0
		2-0CFA							
		3-0CFA							
		1-1CFA	0.997	2,065	796,751	116,593	95.41	339,533	0
		2-2CFA							
		3-3CFA							
		bounded-CPA	1.018	2,069	602,056	311,649	95.41	303,496	0
		SCS	1.019	2,070	602,070	311,649	95.41	303,496	0
		prof (optimisitc)	0.962	1,726	843,031	65,774	95.63	239,678	0

Table B.2: Application performance

	CGC Algorithm	Execution Time	Code Size	# of Calls	# of Sends	% Except Possible	# of Class Tests	# of Heap Alloc
pizza (Java), *without* profile-guided receiver class prediction	none (base)	3.214	2,770	2,811,377	293,038	100.00	1,668,882	0
	selector	2.955	2,507	2,811,377	293,038	98.07	1,668,882	0
	0-Bounded LE	2.992	2,451	2,811,378	275,571	98.06	1,661,512	0
	2-Bounded LE	2.950	2,436	2,811,634	275,291	98.06	1,640,772	0
	4-Bounded LE	2.941	2,422	2,811,988	272,644	98.06	1,639,480	0
	8-Bounded LE	2.977	2,418	2,811,990	272,641	98.06	1,639,482	0
	16-Bounded LE	2.968	2,418	2,811,990	272,641	98.06	1,639,482	0
	32-Bounded LE	2.961	2,418	2,811,990	272,641	98.06	1,639,482	0
	64-Bounded LE	2.968	2,418	2,811,990	272,641	98.06	1,639,482	0
	∞-Bounded LE	2.961	2,405	2,811,992	272,167	98.06	1,639,662	0
	0-Bounded	2.959	2,451	2,811,354	275,571	98.06	1,661,512	0
	2-Bounded	2.964	2,436	2,811,634	275,291	98.06	1,640,690	0
	4-Bounded	2.949	2,418	2,811,990	272,641	98.06	1,639,482	0
	8-Bounded	2.988	2,418	2,811,990	272,641	98.06	1,639,482	0
	16-Bounded	2.947	2,418	2,811,990	272,641	98.06	1,639,482	0
	32-Bounded	2.955	2,418	2,811,990	272,641	98.06	1,639,482	0
	64-Bounded	2.990	2,418	2,811,990	272,641	98.06	1,639,482	0
	∞-Bounded	2.967	2,404	2,811,992	272,167	98.06	1,639,640	0
	0CFA	2.970	2,394	2,811,990	272,172	98.06	1,639,825	0
	1-0CFA	3.219	2,380	2,811,992	272,121	98.06	1,639,644	0
	2-0CFA	3.112	2,380	2,811,992	272,121	98.06	1,639,644	0
	3-0CFA	3.068	2,380	2,811,992	272,121	98.06	1,639,644	0
	1-1CFA	3.240	2,394	2,809,048	271,799	98.09	1,638,088	0
	2-2CFA							
	3-3CFA							
	bounded-CPA	3.001	2,392	2,811,992	272,121	98.06	1,639,644	0
	SCS	2.958	2,392	2,812,665	257,281	98.05	1,644,344	0
	prof (optimisitc)	2.926	2,156	2,828,549	188,455	97.19	1,331,154	0
pizza (Java), *with* profile-guided receiver class prediction	none (base)	3.127	2,775	2,811,353	184,014	100.00	1,796,921	0
	selector	3.041	2,510	2,811,353	184,014	98.01	1,796,921	0
	0-Bounded LE	2.969	2,453	2,811,354	184,003	98.01	1,772,095	0
	2-Bounded LE	2.985	2,438	2,811,634	183,723	98.01	1,751,333	0
	4-Bounded LE	2.957	2,423	2,811,988	181,076	98.00	1,750,117	0
	8-Bounded LE	2.957	2,420	2,811,990	181,073	98.00	1,750,119	0
	16-Bounded LE	2.880	2,420	2,811,990	181,073	98.00	1,750,119	0
	32-Bounded LE	2.883	2,420	2,811,990	181,073	98.00	1,750,119	0
	64-Bounded LE	2.897	2,420	2,811,990	181,073	98.00	1,750,119	0
	∞-Bounded LE	2.957	2,406	2,811,992	180,599	98.00	1,750,299	0
	0-Bounded	2.990	2,453	2,811,354	184,003	98.01	1,772,095	0
	2-Bounded	2.959	2,438	2,811,634	183,723	98.01	1,751,333	0
	4-Bounded	2.935	2,420	2,811,990	181,073	98.00	1,750,119	0
	8-Bounded	2.938	2,420	2,811,990	181,073	98.00	1,750,119	0
	16-Bounded	2.923	2,420	2,811,990	181,073	98.00	1,750,119	0
	32-Bounded	2.909	2,420	2,811,990	181,073	98.00	1,750,119	0
	64-Bounded	2.871	2,420	2,811,990	181,073	98.00	1,750,119	0
	∞-Bounded	2.924	2,406	2,811,992	180,599	98.00	1,750,277	0
	0CFA	2.908	2,394	2,811,992	180,599	98.00	1,750,277	0
	1-0CFA	2.925	2,381	2,811,992	180,553	98.00	1,750,281	0
	2-0CFA	2.951	2,381	2,811,992	180,553	98.00	1,750,281	0
	3-0CFA	2.948	2,381	2,811,992	180,553	98.00	1,750,281	0
	1-1CFA	3.249	2,395	2,809,024	180,494	98.03	1,748,472	0
	2-2CFA							
	3-3CFA							
	bounded-CPA	2.981	2,393	2,811,992	180,553	98.00	1,750,281	0
	SCS	2.929	2,393	2,812,665	175,141	98.00	1,745,015	0
	prof (optimisitc)	2.821	2,158	2,828,549	83,175	97.56	1,639,062	0

Table B.2: Application performance

	CGC Algorithm	Execution Time	Code Size	# of Calls	# of Sends	% Except Possible	# of Class Tests	# of Heap Alloc
richards (Smalltalk) *without* profile-guided receiver class prediction	none (base)	0.209	2,954	305,096	417,809	100.00	4,562,001	1,152
	selector	0.215	2,760	305,096	417,809	90.58	4,561,488	1,151
	0-Bounded LE	0.215	2,759	305,096	417,809	90.58	4,561,484	1,151
	2-Bounded LE	0.205	2,629	305,096	415,823	90.61	4,556,604	1,149
	4-Bounded LE	0.221	2,579	305,096	413,496	90.58	4,413,151	1,149
	8-Bounded LE	0.201	2,579	305,096	413,496	90.58	4,413,151	1,149
	16-Bounded LE	0.202	2,579	305,096	413,496	90.58	4,413,151	1,149
	32-Bounded LE	0.202	2,579	305,096	413,496	90.58	4,413,151	1,149
	64-Bounded LE	0.209	2,580	305,096	413,496	90.58	4,413,151	1,149
	∞-Bounded LE	0.190	2,546	305,096	411,478	90.55	4,310,481	1,149
	0-Bounded	0.214	2,759	305,096	417,809	90.58	4,561,484	1,151
	2-Bounded	0.206	2,635	305,096	415,823	90.61	4,556,348	1,149
	4-Bounded	0.211	2,589	305,096	413,496	90.58	4,412,895	1,149
	8-Bounded	0.197	2,590	305,096	413,496	90.58	4,412,895	1,149
	16-Bounded	0.199	2,590	305,096	413,496	90.58	4,412,895	1,149
	32-Bounded	0.199	2,590	305,096	413,496	90.58	4,412,895	1,149
	64-Bounded	0.210	2,591	305,096	413,496	90.58	4,412,895	1,149
	∞-Bounded	0.197	2,529	305,096	411,478	90.55	4,310,461	1,149
	0CFA	0.201	2,055	305,096	403,506	90.45	4,186,092	14
	1-0CFA							
	2-0CFA							
	3-0CFA							
	1-1CFA							
	2-2CFA							
	3-3CFA							
	bounded-CPA							
	SCS							
	prof (optimisitc)	0.190	1,112	307,170	210,264	98.75	2,633,761	11
richards (Smalltalk) *with* profile-guided receiver class prediction	none (base)	0.156	4,299	305,096	209,036	100.00	3,955,653	1,152
	selector	0.172	4,060	305,096	209,036	99.61	3,955,140	1,151
	0-Bounded LE	0.176	4,060	305,096	209,036	99.61	3,955,136	1,151
	2-Bounded LE	0.158	3,718	305,096	208,929	99.61	3,949,839	1,149
	4-Bounded LE	0.167	3,657	305,096	208,765	99.61	3,884,131	1,149
	8-Bounded LE	0.158	3,657	305,096	208,765	99.61	3,884,131	1,149
	16-Bounded LE	0.154	3,657	305,096	208,765	99.61	3,884,131	1,149
	32-Bounded LE	0.154	3,657	305,096	208,765	99.61	3,884,131	1,149
	64-Bounded LE	0.162	3,659	305,096	208,765	99.61	3,884,131	1,149
	∞-Bounded LE	0.170	3,607	305,096	208,513	99.62	3,791,450	1,149
	0-Bounded	0.177	4,060	305,096	209,036	99.61	3,955,136	1,151
	2-Bounded	0.160	3,719	305,096	208,929	99.61	3,949,583	1,149
	4-Bounded	0.193	3,666	305,096	208,765	99.61	3,883,875	1,149
	8-Bounded	0.169	3,666	305,096	208,765	99.61	3,883,875	1,149
	16-Bounded	0.165	3,666	305,096	208,765	99.61	3,883,875	1,149
	32-Bounded	0.164	3,666	305,096	208,765	99.61	3,883,875	1,149
	64-Bounded	0.171	3,668	305,096	208,765	99.61	3,883,875	1,149
	∞-Bounded	0.163	3,591	305,096	208,513	99.62	3,791,430	1,149
	0CFA	0.151	2,154	305,712	206,928	99.62	3,735,425	12
	1-0CFA							
	2-0CFA							
	3-0CFA							
	1-1CFA							
	2-2CFA							
	3-3CFA							
	bounded-CPA							
	SCS							
	prof (optimisitc)	0.150	1,108	307,170	200,673	99.66	2,478,050	11

Table B.2: Application performance

Benchmark	Configuration	CGC Algorithm	Execution Time	Code Size	# of Calls	# of Sends	% Except Possible	# of Class Tests	# of Heap Alloc
deltablue (Smalltalk)	*without* profile-guided receiver class prediction	none (base)	0.055	2,984	428,927	14,475,305	100.00	22,680,348	139,472
		selector	0.054	2,802	428,927	14,475,305	98.10	22,679,835	139,471
		0-Bounded LE	0.053	2,801	428,927	14,475,305	98.10	22,679,833	139,471
		2-Bounded LE	0.055	2,678	428,929	14,473,319	98.11	22,674,771	139,471
		4-Bounded LE	0.054	2,641	428,929	14,473,319	98.11	22,581,368	139,471
		8-Bounded LE	0.053	2,641	428,929	14,473,319	98.11	22,581,368	139,471
		16-Bounded LE	0.054	2,641	428,929	14,473,319	98.11	22,581,368	139,471
		32-Bounded LE	0.055	2,641	428,929	14,473,319	98.11	22,581,368	139,471
		64-Bounded LE	0.056	2,642	428,929	14,473,319	98.11	22,581,368	139,471
		∞-Bounded LE	0.060	2,606	429,029	14,470,693	98.11	22,423,638	139,471
		0-Bounded	0.056	2,801	428,927	14,475,305	98.10	22,679,833	139,471
		2-Bounded	0.057	2,684	428,929	14,473,319	98.11	22,674,515	139,471
		4-Bounded	0.054	2,651	428,929	14,473,319	98.11	22,581,112	139,471
		8-Bounded	0.057	2,651	428,929	14,473,319	98.11	22,581,112	139,471
		16-Bounded	0.058	2,651	428,929	14,473,319	98.11	22,581,112	139,471
		32-Bounded	0.058	2,651	428,929	14,473,319	98.11	22,581,112	139,471
		64-Bounded	0.058	2,653	428,929	14,473,319	98.11	22,581,112	139,471
		∞-Bounded	0.059	2,589	429,029	14,470,693	98.11	22,423,214	139,471
		0CFA	0.050	2,141	434,129	14,245,787	98.08	22,455,941	20,608
		1-0CFA							
		2-0CFA							
		3-0CFA							
		1-1CFA							
		2-2CFA							
		3-3CFA							
		bounded-CPA							
		SCS							
		prof (optimisitc)	0.034	1,171	452,610	5,861,542	97.27	19,988,318	20,605
	with profile-guided receiver class prediction	none (base)	0.046	3,693	433,930	10,713,261	100.00	24,020,274	139,472
		selector	0.045	3,474	433,930	10,713,261	99.98	24,019,761	139,471
		0-Bounded LE	0.045	3,473	433,930	10,713,261	99.98	24,019,759	139,471
		2-Bounded LE	0.047	3,172	433,932	10,713,052	99.98	24,014,283	139,471
		4-Bounded LE	0.050	3,124	433,932	10,712,786	99.98	24,014,365	139,471
		8-Bounded LE	0.047	3,123	433,932	10,712,786	99.98	24,014,365	139,471
		16-Bounded LE	0.045	3,123	433,932	10,712,786	99.98	24,014,365	139,471
		32-Bounded LE	0.046	3,123	433,932	10,712,786	99.98	24,014,365	139,471
		64-Bounded LE	0.045	3,127	433,932	10,712,786	99.98	24,014,365	139,471
		∞-Bounded LE	0.046	3,084	434,032	10,712,226	99.98	23,857,220	139,471
		0-Bounded	0.048	3,473	433,930	10,713,261	99.98	24,019,759	139,471
		2-Bounded	0.046	3,192	433,932	10,713,052	99.98	24,014,027	139,471
		4-Bounded	0.047	3,149	433,932	10,712,786	99.98	24,014,109	139,471
		8-Bounded	0.047	3,150	433,932	10,712,786	99.98	24,014,109	139,471
		16-Bounded	0.047	3,150	433,932	10,712,786	99.98	24,014,109	139,471
		32-Bounded	0.047	3,150	433,932	10,712,786	99.98	24,014,109	139,471
		64-Bounded	0.045	3,153	433,932	10,712,786	99.98	24,014,109	139,471
		∞-Bounded	0.047	3,067	434,032	10,712,226	99.98	23,856,796	139,471
		0CFA	0.045	2,324	434,737	11,208,647	98.69	24,508,050	20,608
		1-0CFA							
		2-0CFA							
		3-0CFA							
		1-1CFA							
		2-2CFA							
		3-3CFA							
		bounded-CPA							
		SCS							
		prof (optimisitc)	0.035	1,165	452,610	5,594,225	99.95	19,774,573	20,605

Appendix C

Relative Impact of Intraprocedural Analyses

The experimental results of chapter 4 aggregated five distinct interprocedural analyses into a single IP configuration. This appendix examines the relative contribution to the overall improvement in bottom-line application performance made by each individual analysis. The basic methodology will be to compare the performance of twelve configurations: the **base** and **base+IP** configurations of chapter 4, five configurations that augment **base** with only one interprocedural analysis, and five configurations that augment **base** with all but one of the five interprocedural analyses. To reduce the number of experiments to a manageable number, only two programs, instr sched and cassowary, and one call graph construction algorithm, SCS, will be considered. Both of these programs are moderately sized and obtained relatively large speedups in the **base+IP**$_{\mathbf{SCS}}$ configuration.

Figure C.1 presents the results of these experiments. Normalized execution speed is shown for each of the twelve configurations; the two horizontal lines indicate the speed of the **base** and **base+IP**$_{\mathbf{SCS}}$ configurations. For both programs, interprocedural class analysis (*class*) is clearly responsible for virtually all of the performance difference between **base** and **base+IP**$_{\mathbf{SCS}}$. There are variations in the performance of the other configurations, however the differences are too small to be conclusive. The other analyses are enabling additional optimizations, but the absolute impact of these optimizations is negligible. For example, the **base+MOD-only** configuration of instr sched executes 1.5% fewer instance variable loads than the **base** configuration: by enabling Vortex to keep more intraprocedural information alive across calls, MOD analysis

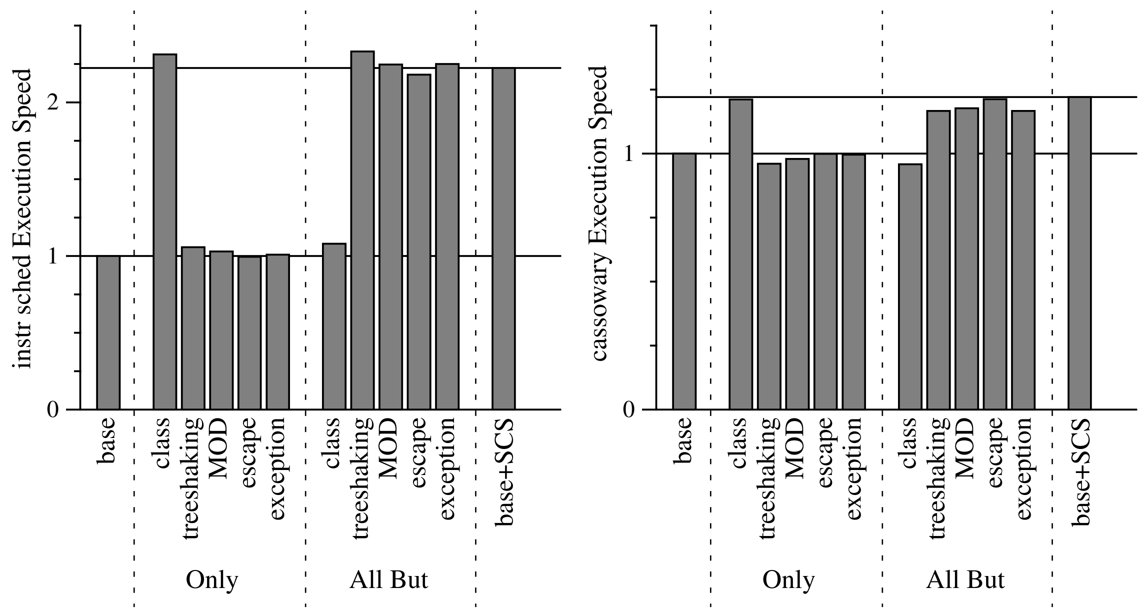


Figure C.1: Performance impact of individual interprocedural analyses

improves the effectiveness of Vortex's redundant load elimination optimization. As explained in section B.2, escape analysis was not important for instr sched due to Vortex's exploitation of the non-escaping annotation.

Vita

David Paul Grove was born on April 2, 1970 in Pittsburgh, PA. Soon afterwards, he moved to the suburbs of Boston where he remained until Fall 1988 when he matriculated at Yale College. He attended Yale College until Spring 1992, graduating *magna cum laude with distinction in Computer Science* with a B.S. in Computer Science. He was a graduate student in the University of Washington Department of Computer Science and Engineering from Fall 1992 through Fall 1998 earning his M.S. in Computer Science in 1994 and his Ph.D. in 1998.