A Framework for Call Graph Construction Algorithms

DAVID GROVE
IBM T.J. Watson Research Center
and
CRAIG CHAMBERS
University of Washington, Seattle

A large number of call graph construction algorithms for object-oriented and functional languages have been proposed, each embodying different tradeoffs between analysis cost and call graph precision. In this article we present a unifying framework for understanding call graph construction algorithms and an empirical comparison of a representative set of algorithms. We first present a general parameterized algorithm that encompasses many well-known and novel call graph construction algorithms. We have implemented this general algorithm in the Vortex compiler infrastructure, a mature, multilanguage, 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. For many of these 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.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*applicative* (*functional*) *languages*; *object-oriented languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Classes and objects*; *procedures, functions, and Subroutines*; D.3.4 [**Programming Languages**]: Processors—*compilers*; *optimization*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Call graph construction, control flow analysis, interprocedural analysis

1. INTRODUCTION

Frequent procedure calls and message sends serve as important structuring tools for object-oriented languages; but they can also severely degrade application runtime performance. This degradation is due to both the direct cost of implementing the call and to the indirect cost of missed opportunities for

Authors' addresses: D. Grove, IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598; email: groved@us.ibm.com; C. Chambers, Dept. of Computer Science and Engineering, University of Washington, P.O. Box 352350, Seattle, WA 98195; email: chambers@cs.washington.edu

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 calls, thus removing both the direct and indirect costs of the call. However, in most programs, even after these techniques have been applied, there will be some remaining nonstatically bound message sends and noninlined procedure calls. If the presence of these call sites forces an optimizing compiler to make overly pessimistic assumptions, then potentially profitable optimization opportunities can 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 noninlined calls, thus enabling it to make less pessimistic assumptions about program behavior and reduce the performance impact of noninlined call sites. An interprocedural analysis can be divided into two logically separate subtasks. First, the *program call graph*, a compile-time data structure that represents the runtime calling relationships among a program's procedures, is constructed. In most cases this is done as an explicit prephase before performing the "real" interprocedural analysis, although some analyses 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 and 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 is determined by the function value itself. 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 (collectively called "higher-order" languages) and interprocedural analysis of strictly first-order procedural languages. Effectively resolving this circularity is the primary challenge of the call graph construction problem for higher-order languages.

The main contributions of this work are the following:

—We developed a general parameterized algorithm for call graph construction. 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 algorithmic design space. The general algorithm is quite expressive, encompassing algorithms with a wide range of cost and precision characteristics.

—We implemented the general algorithm in the Vortex compiler infrastructure. The definition 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 within a mature optimizing compiler provides a "level playing field" for meaningful cross-algorithm performance comparisons.

—We experimentally assessed a representative selection of call graph construction algorithms. The Vortex implementation framework was used to test each algorithm on a suite of sizeable (5,000 to 50,000 lines of code) Cecil and Java programs. In comparison to previous experimental studies, our experiments cover a much wider range of algorithms and include applications that are an order of magnitude larger than the largest used in prior work. Assessing the algorithms on larger programs is important because some of the algorithms, including one that was claimed scalable in previous work cannot be practically applied to some of our larger benchmarks.

This article is organized into two major parts: a presentation of our general parameterized call graph construction algorithm (Sections 2 through 5) and an empirical assessment of a representative set of instantiations of the general algorithm (Sections 6 through 11). Section 2 begins by reviewing the role of class analysis in call graph construction. Sections 3 and 4 are the core of the first half of the article; they formally define call graphs and the call graph construction problem and present our general parameterized algorithm. Section 5 describes our implementation of the algorithm in the Vortex compiler infrastructure: it focuses on our design choices and how key primitives are implemented. Section 6 begins the second half of the article by describing our experimental methodology. Sections 7 through 10 each describe and empirically evaluate a category of call graph construction algorithms. Section 11 concludes the second half of the article by summarizing our experimental results. Section 11 also uses the insights gained from defining each algorithm as an instance of our general algorithm to partially order algorithms based on the relative precision of the call graphs they compute. Finally, Section 12 discusses some additional related work and Section 13 concludes.

2. 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 call site source code. The problem is that the target methods that will be invoked as a result of the message send are determined by the classes of the objects that reach the message send site at runtime, and thus act as receivers for the message. In general, the flow of objects to call sites may be interprocedural, and thus precisely determining the receiver class sets 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. There are two possible approaches for handling the circular dependencies among the receiver class sets, the program call graph, and interprocedural analysis:

—*We can make 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. 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. This overall process can be repeated iteratively 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.

—*We can make an optimistic (but likely unsound) assumption*, and then iterate to a sound fixed-point solution. Just as 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 fixed 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[1] 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 requiring further analysis. The optimistic approach can yield more precise call graphs (and receiver class sets) than the pessimistic one, but is more complicated and may be more computationally expensive.

This article presents and empirically assesses call graph construction algorithms that utilize both 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 differences in precision impact the effectiveness of client interprocedural analyses, and in turn substantially impact bottom-line application performance. Therefore, Sections 3 and 4 concentrate on developing formalisms that naturally support describing optimistic

[1] We define the *main routine* of a program as the union of all program entry points and static data initializations.

algorithms that integrate call graph construction and interprocedural class analysis (although the formalisms can also describe some pessimistic algorithms).

3. 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 3.1 informally introduces some of the key ideas. Section 3.2 formalizes the intuition of Section 3.1 using ideas drawn from lattice theory. Finally, Section 3.3 discusses the termination and soundness of call graph construction algorithms.

3.1 Informal Model of Call Graphs

In its standard formulation, a call graph consists of nodes, representing procedures, linked by directed edges, representing calls from one procedure to another. However, this formulation is insufficient to accurately capture the output of a context-sensitive interprocedural class analysis algorithm. Instead, each call graph node will represent a *contour*: an analysis-time representation of a procedure. In context-insensitive call graphs, there is exactly one contour for each procedure; in context-sensitive call graphs, there may be an arbitrary number of contours representing different analysis-time views of a single procedure. Figure 1(b) shows the context-insensitive call graph[2] for the example program from Figure 1(a). Figure 1(c) depicts the call graph constructed by a context-sensitive algorithm that has created additional contours to separate the flow of `Circle` and `Square` objects from their creation points in methods `A` and `B` to their use as receivers of the `area` message in the `sumArea` method.

Abstractly, every contour contains three components: an *identifier* that encodes which program construct the contour represents; a *contour key* that encodes the contexts in which the contour applies; and *analysis information* to associate with the program construct in the contexts encoded by the contour key. Typically, the program constructs represented by contours are the procedures of a program. However, as described below, contours can also be used to represent classes and instance variables to obtain context-sensitive analysis of polymorphic data structures. In the example program, determining that `sumArea` called from `A` can only invoke the `area` method of the `Circle` class (and cannot invoke `Square::area`) and requires the use of class and instance variable contours to more precisely represent the flow of values through the `first` and `second` instance variables of the `SPair` class.

For the purposes of interprocedural class analysis, a *procedure contour* contains the following analysis information

—*Class sets* represent the result of interprocedural class analysis. Every contour contains class sets for formal parameters, local variables, and the result of the procedure. These sets of classes represent the possible classes of objects stored in the corresponding variable (or returned from the procedure)

[2] To simplify the figure, all calls to constructors are omitted.

```
class Shape {
   abstract float area();
}

class Square extends Shape {
   float size;
   Square(float s) {
      size = s;
   }
   float area() {
      return size * size;
   }
}

class Circle extends Shape {
   float radius;
   Circle(float r) {
      radius = r;
   }
   float area() {
      return PI*radius*radius;
   }
}

class SPair {
   Shape first;
   Shape second;
   SPair(Shape s1, Shape s2) {
      first = s1; second = s2;
   }
}
```

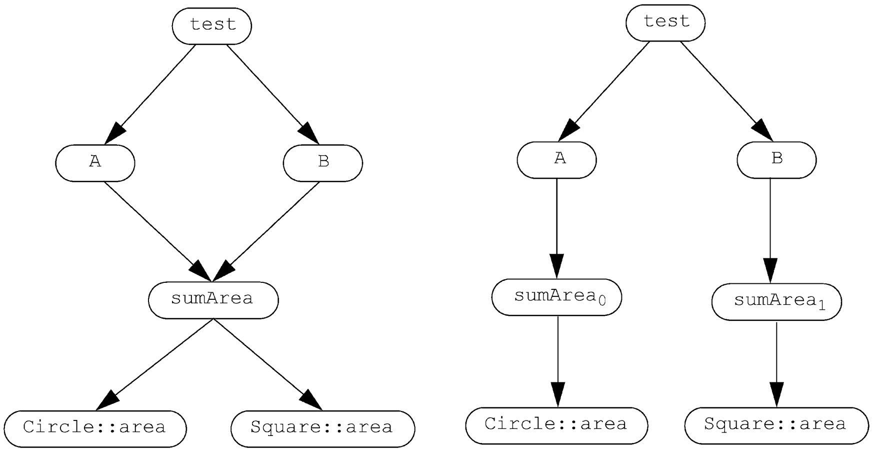
```
class Example {
   float test(float v1, float v2) {
      return A(v1, v2) + B(v1, v2);
   }

   float A(float v1, float v2) {
      Circle c1 = new Circle(v1);
      Circle c2 = new Circle(v2);
      return sumArea(new SPair(c1, c2));
   }

   float B(float v1, float v2) {
      Square s1 = new Square(v1);
      Square s2 = new Square(v2);
      return sumArea(new SPair(s1, s2));
   }

   float sumArea(SPair p) {
      return p.first.area() + p.second.area();
   }
}
```

(a) Example program fragment



(b) Context-Insensitive

(c) Context-Sensitive

Fig. 1. Context-insensitive vs. context-sensitive call graph.

during program execution. If the analysis has computed that an instance of some class `C` could be stored in a program variable `x`, then the class set for `x` will contain `C`. If some subclass of `C`, `D` is not contained in the set, then the analysis has determined that instances of `D` cannot be stored in `x`.

—*Call graph edges* record for each call site a set of possible callee contours.

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. An *instance variable contour* associates a single class set representing the potential classes of values stored in the instance variable with an instance variable identifier and contour key pair.

To more precisely analyze polymorphic data structures, including array classes, some interprocedural class analysis algorithms introduce additional context sensitivity in their analysis of classes and instance variable content. 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*. A class contour does not contain any additional analysis information, and thus consists of just a class identifier and a contour key. In the literature, using multiple class contours is sometimes described as tagging classes with their *creation points*. All previously described class information, for example the class sets recorded in procedure and instance variable contours, is generalized to be class contour information.

One of the most commonly used forms of context sensitivity is to use a vector of the k enclosing calling procedures as a contour key (the "call strings" approach of Sharir and Pnueli [1981]). Using call strings with $k = 1$ as the contour keys for procedure and class contours and using the class contours themselves as the contour keys for instance variable contours, would define one (of many possible) context-sensitive call graph construction algorithm that would compute the call graph of Figure 1(c). For this example, the key to obtaining a precise call graph is to accurately model the flow of values through the instance variables of the `SPair` class. The analysis creates two class contours $\langle SPair, A\rangle$ and $\langle SPair, B\rangle$ to represent `SPair` instances created in `A` and `B`, respectively. Instance variable contours for `first` and `second` are created for each class contour. The class sets of the instance variable contours with the key $\langle SPair, A\rangle$ contain only `Circle`; those of $\langle SPair, B\rangle$ contain only `Square`. The class set of the formal parameter `p` of contour $\texttt{SumArea}_0$ (which represents `sumArea` called from `A`) contains only $\langle SPair, A\rangle$, and thus the analysis can determine that only `Circle::area` could be invoked from $\texttt{SumArea}_0$.

The global scope can be represented by a special "root" procedure contour: its local variables are the program's global variables and its body is the program's main routine. Since, unlike a procedure, the global scope cannot be analyzed context-sensitively, exactly one contour is created to represent it.

To analyze languages with lexically nested functions, we can augment procedure contours with a lexical parent contour key that encodes the contours' lexical nesting relationship. This information is used to allow contours representing lexically nested procedures to access the class sets of free variables from the appropriate contour(s) of the enclosing procedure.

Informally, the result of the combined call graph construction and interprocedural class analysis algorithm, defined in Section 4, is a set of procedure contours and a set of instance variable contours. Together, the content of these two sets define a contour call graph (the call graph edges component of the procedure contours) and class contour sets for all interesting program constructs (the class set component of the procedure and instance variable contours).

3.2 Formal Model of Call Graphs

This section uses lattice-theoretic ideas to formally define the contour-based model of context-sensitive call graphs. 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 the pair of a particular input program and a particular call graph construction algorithm forms a downward semilattice; we sometimes use the term "domain" as shorthand for a downward semilattice. As is traditional in dataflow analysis [Kildall 1973; Kam and Ullman 1976] (but opposite to the conventions in programming language semantics and abstract interpretation [Cousot and Cousot 1977]), 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, $\bot$, 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 3.3.2 formally defines soundness and some of the related structures of call graph domains.

3.2.1 *Supporting Domain Constructors.*

The definition of the call graph domain uses several auxiliary domain constructors to abstract common patterns, thus 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 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 $S_{DPS} = \{Bottoms(S) | S \in PowerSet(S_D)\}$, where $Bottoms(S) = \{d \in S | \neg(\exists d' \in S, d' \leq_D d)\}$. 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* differs subtly from the standard powerset domain constructor, which maps an unordered input set to a lattice whose domain is the full powerset of its input with a partial order based solely on the subset relationship. The more complex definition of *Pow* preserves the relationships established by its input partial order. For example, if $a{\leq_D}b$ then $\{a\}{\leq_{pow(D)}}\{b\}$, but under the standard powerset domain constructor, $\{a\}$ and $\{b\}$ are unordered. $Bottoms(S)$ removes those elements that are redundant with respect to $\leq_D$ from S.

The constructor *Map* is a function constructor that 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 $S_M = \{f \subseteq X \times S_Y \mid (x, y_1) \in f \wedge (x, y_2) \in f \Rightarrow y_1 = y_2\}$ 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

$$GLB_1 = \{(x, y) \mid (x, y_1) \in m_1, (x, y_2) \in m_2, y = glb_Y(y_1, y_2)\}$$
$$GLB_2 = \{(x, y) \mid (x, y) \in m_1, x \notin dom(m_2)\}$$
$$GLB_3 = \{(x, y) \mid (x, y) \in m_2, x \notin dom(m_1)\}$$

The constructor *Union* takes two input partial orders $D_1 = \langle S_1, \leq_1 \rangle$ and $D_2 = \langle S_2, \leq_2 \rangle$ where $S_1 \cap S_2 = \emptyset$, and generates a new partial order $U = \langle S_U, \leq_U \rangle$ where $S_U = S_1 \cup S_2$ and $a{\leq_U}b \Leftrightarrow a{\leq_1}b \vee a{\leq_2}b$. If the input partial orders are downward semilattices, then U is also a downward semilattice. If two elements of S_U are drawn from the same input partial order, their greatest lower bound is the greatest lower bound from that partial order; if they are drawn from different input partial orders then their greatest lower bound is $\perp$.

Each member of the family of constructors *kTuple*, $\forall k \geq 0$, is the standard k-tuple constructor that 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.

Finally, the constructor *Seq* 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 $1Tuple(D)$, all of, $2Tuple(D, D)$, all of $3Tuple(D, D, D)$, and so on, and the partial order $\leq_V$ is the union of the individual k-tuple partial orders with the partial order $\{(\perp, e) \mid e \in S_V\}$. 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 *Call Graph Domain.* Figure 3 utilizes the domain constructors specified in the previous section to define the call graph domain for each pair of a particular input program and a particular call graph construction algorithm. This definition is parameterized by the sets and partial orders listed in Figure 2 that encode program features and algorithm-specific context-sensitivity polices. The ordering relation on the *ProcKey*, *InstVarKey*, and *ClassKey* partial orders (and all derived domains) indicates the relative precision of the elements: one

Features of a given program are abstracted into seven unordered sets

Class	all class declarations of the program
InstVariable	all instance variable declarations of the program
Procedure	all procedure declarations of the program
Variable	all variable names used in the program
CallSite	all call sites in the program
NewSite	all object instantiation sites in the program
LoadSite	all loads of instance variables in the program
StoreSite	all stores to instance variables in the program

The context-sensitivity decisions of the call graph construction algorithm are encoded by three partial orders

ProcKey defines the space of possible contexts for context-sensitive analysis of functions by serving as the contour key for procedure contours.

InstVarKey defines the space of possible contexts for separately tracking the contents of instance variables by serving as the contour key for instance variable contours.

ClassKey defines the space of possible contexts for context-sensitive analysis of classes by serving as the contour key for class contours.

Fig. 2. Parameters of the call graph domain construction (see Figure 3).

CallGraph = 2Tuple(InstVarContourMap, ProcContourMap)

InstVarContourMap= Map(InstVarID, InstVarContour)

InstVarContour = 2Tuple(InstVarID, ClassContourSet)

InstVarID = 2Tuple(InstVariable, InstVarKey)

ProcContourMap = Map(ProcID, ProcContour)

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

ProcID = 3Tuple(Procedure, ProcKey, Pow(Seq(ProcKey)))

ClassContourSet = Pow(ClassContour)

ClassContour = 2Tuple(Class, Union(ClassKey, Seq(ProcKey)))

Fig. 3. Construction of the *CallGraph* domain for a program and call graph construction algorithm pair.

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 maps describing the instance variable contours and procedure contours contained in the call graph. 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. 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 contours' *ProcIDs* is restricted to sets of sequences of exactly length n, where n is the lexical nesting depth of the procedure. *InstVarContourMap* and *ProcContourMap* are constrained such that for each key k, the contour that k is mapped to has k as its first component.

—The second component of both the 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 intercontour 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 contains an element of the *ClassKey* domain. For closures, the second component contains a sequence of *ProcKeys* that encode the lexical chain of procedure contours that should be used to analyze any references to free variables contained in the closure. This lexical chain information is used when the closure is invoked and a procedure contour is created for the closure to initialize the third component of its *ProcID*.

For example, the 0-CFA algorithm is the classic context-insensitive call graph construction algorithm for object-oriented and functional languages [Shivers 1988; 1991]. It can be modeled by using the single-point lattice, $\{\bot\}$, to instantiate the *ProcKey*, *InstVarKey*, and *ClassKey* partial orders. Thus, each call graph has 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 extended 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.

3.3 Properties

3.3.1 *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). 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 domain 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 $O(\textit{LatticeHeight} \times \textit{StepCost})$.

All of the sets specifying program features are 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 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* be a set of sequences of a length that exactly matches the lexical nesting depth of their procedure, and the absence of any mutually recursive equations in Figure 3. 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 and Cousot 1977] to ensure termination. For example, the Cartesian product [Agesen 1995] and SCS [Grove et al. 1997] algorithms, described in Section 9.1.2, both use elements of the *ClassContour* domain as part of their *ProcKey* domain elements. In the presence of closures (represented by the analysis as class contours), this can lead to an infinitely tall call graph domain when 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 1996].

3.3.2 *Soundness.* Figure 4 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., $B \leq_{cg} A$, then A will be located in the cone above B in the diagram, and likewise B will be located in the cone below A. 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, hence 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

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

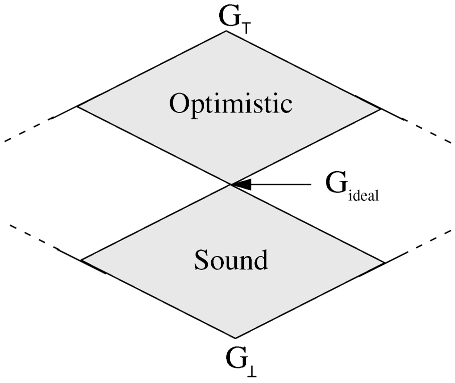


Fig. 4. Regions in a call graph domain.

test for the soundness of G. Note that not all call graphs are ordered with respect to G_{ideal}; Figure 4 only depicts a subset of the elements of a call graph domain.

4. A PARAMETERIZED CALL GRAPH CONSTRUCTION ALGORITHM

This section specifies our general integrated call graph construction and interprocedural class analysis algorithm for the small example language defined in Section 4.1. 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 4.2. Section 4.3 defines additional notation and auxiliary functions. Section 4.4 specifies the analysis performed by the algorithm using set constraints, and Section 4.5 discusses methods of constraint satisfaction.

4.1 A Simple Object-Oriented Language

The analysis is defined on the simple statically typed object-oriented language whose abstract syntax is given by Figure 5.[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 a unified subtyping and inheritance hierarchy can be modeled by requiring that all subtyping and inheritance declarations be parallel. Multimethods generalize the singly-dispatched methods found in many object-oriented 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

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

```
Program      ::= {TopDecl} {Stmt} Expr
TopDecl      ::= TypeDecl | ClassDecl | 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 | IVAssign
VarAssign    ::= VarID := Expr
IVAssign     ::= Expr . InstVarID := Expr
Expr         ::= VarRef | IVRef | NewExpr | ClosureExpr |
                  SendExpr | ApplyExpr
VarRef       ::= VarID | FormalID
IVRef        ::= Expr . InstVarID
NewExpr      ::= new ClassID
ClosureExpr ::= lambda ({Formal}):TypeID { {VarDecl} {Stmt} Expr }
SendExpr     ::= send MsgID ( {Expr} )
ApplyExpr    ::= apply Expr ( {Expr} )
```

Fig. 5. Abstract syntax for simple object-oriented language.

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 one (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 (i.e., by 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 is 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, 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 nonlocal 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 Java, with the exception of 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 that is independent of program size and that the static number

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):

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

Environment Key Selection Function (EKS):

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

Fig. 6. Signatures of contour key selection functions.

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.

4.2 Algorithm Parameters

The general algorithm is parameterized by four *contour key selection functions* that collaborate to define the context-sensitivity policies 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, which 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 6. 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 (sequences of) 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 function follow:

—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. If the call site has multiple possible callee procedures, then the analysis will select contour keys for each in turn by invoking the PKS function once for each potential callee.

—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 whose instance variable is being accessed). It returns a set of instance variable contour keys.

—The class contour key selection function (CKS) determines the class contours that should be created to represent the objects created at class instantiation sites. Its arguments are the class being instantiated, the instantiation site, 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 reference to free variables contained in a closure. Its arguments are the closure being instantiated and the procedure contour in which the instantiation is analyzed. It returns a set of sequences 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 in Figure 8).

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 nonempty. Contour key selection functions return sets of contour keys (rather than just a single contour key) to enable algorithms to decompose the analysis of a program construct in a particular context into multiple cases, each handled by a different contour.

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 4.3). 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 4.5).

4.3 Notation and Auxiliary Functions

This section defines the notational conventions and auxiliary functions in the algorithm specification of Figure 9.

During analysis, sets of class contours are associated with every expression, variable (including formal parameters), and instance variable in the program.

[5] A function F is monotonic iff $x \leq y \Rightarrow F(x) \leq F(y)$.

The class contour set associated with the program construct PC in contour κ is denoted $[\![\mathrm{PC}]\!]_\kappa$. 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 judgments of the form $\kappa \vdash \mathrm{PC} \Rightarrow C$, which should be read as the analysis of program construct PC in the context of procedure contour κ gives rise to the constraint set C. These judgments are combined in inference rules, which can be understood informally as inductively defining the analysis of a program construct in terms of the analysis of its subcomponents. For example, the [Seq] rule in Figure 9 describes the analysis of a sequence of statements in terms of the analysis of individual statements:

$$\frac{\begin{array}{c}\kappa \vdash \mathrm{s}_1 \Rightarrow C_1 \\ \kappa \vdash \mathrm{s}_2 \Rightarrow C_2\end{array}}{\kappa \vdash \mathrm{s}_1 ; \mathrm{s}_2 \Rightarrow C_1 \wedge C_2}$$

To analyze the program construct $\mathtt{S}_1;\mathtt{S}_2$, 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 nonnegative 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 subsystem may perform to satisfy the constraint. Section 5.4 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 a pure equality constraint and sets are unified as soon as the constraint is created between them. Finally, if $p = \infty$, then the generalized constraint is a pure 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 which arise from static type declarations and method specializers.

A number of auxiliary functions are used to concisely specify the analysis. Figure 7 lists auxiliary functions for manipulating program constructs and accessing components of the call graph domain (defined in Figure 3). Figure 8 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 used to analyze a reference to variable V made from procedure contour K.

Functions for manipulating program constructs:

Type(`PC`*)*	returns the static type of `PC`
FormalDecl(*i*, `M`)	returns *i*-th formal of method `M`
Body(`M`)	returns the body of method `M`
Conformers(`T`)	the set of classes that subtype `T` (from class hierarchy analysis)
Subclasses(`C`)	the set of classes that inherit from `C` (from class hierarchy analysis)
LexEnclProc(`M`)	the lexically enclosing method of method `M`

Functions for accessing components of k-tuples and maps:

If $id = (a, b) \in InstVarID$ then $InstVar(id) = a$ and $Key(id) = b$.

If $id = (a, b, c) \in ProcID$ then $Proc(id) = a$, $Key(id) = b$, and $Lex(id) = c$.

If $\kappa = (a, b) \in InstVarContour$ then $ID(\kappa) = a$ and $Contents(\kappa) = b$.

If $\kappa = (a, b, c, d, e) \in ProcContour$ then $ID(\kappa) = a$, $VarMap(\kappa) = b$, $CallSites(\kappa) = c$. $LoadSites(\kappa) = d$, and $StoreSites(\kappa) = e$.

We define several shorthands for accessing the codomain of b: $Var(\mathbf{V}, \kappa) = b(\mathbf{V})$, $Result(\kappa) = b(\mathtt{result})$, and $Formal(i, \kappa) = b(FormalDecl(i, Proc(a)))$.

If $\kappa = (a, b) \in ClassContour$ then $Class(\kappa) = a$. and $Key(\kappa) = b$.

Fig. 7. Named k-tuple and map accessors for call graph domain constituents.

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

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

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

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

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

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

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

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

Fig. 8. Auxiliary functions.

—*ICVS* determines the target contours for an instance variable load or store on the basis of a 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 nests all methods in the global scope. In contrast, *ACS* must extract the set of lexical chains from the second component of the closure class contours.

4.4 Algorithm Specification

Figure 9 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).

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 for explicit assignments, [VAssign] and [IVarAssign], 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 aimed at assignments can still be beneficial because some algorithm instantiations may not be as precise as the language static type system.

The [Program] rule is the entry to the analysis; the top-level statements and expression are analyzed in the context of κ_{root}, the contour representing 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, $[\![\mathrm{E}]\!]_{\kappa}$, 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.

Basic expressions are handled by the next four rules. Variable and instance variable references use their respective auxiliary functions to find a set of target

[Program]

$$\frac{\begin{array}{c}\kappa_{root} \vdash \mathtt{S} \Rightarrow C_1 \\ \kappa_{root} \vdash \mathtt{E} \Rightarrow C_2\end{array}}{\vdash \mathtt{D\ S\ E} \Rightarrow C_1 \wedge C_2}$$

[Seq]

$$\frac{\begin{array}{c}\kappa \vdash \mathtt{S_1} \Rightarrow C_1 \\ \kappa \vdash \mathtt{S_2} \Rightarrow C_2\end{array}}{\kappa \vdash \mathtt{S_1;S_2} \Rightarrow C_1 \wedge C_2}$$

[VarAssign]

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

[IVarAssign]

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

[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]

$$\kappa \vdash \mathtt{new}_l\ \mathtt{C} \Rightarrow [\![\mathtt{new}_l\ \mathtt{C}]\!]_\kappa \supseteq_p \{\langle \mathtt{C}, key\rangle \mid key \in CKS(\mathtt{C}, l, \kappa)\}$$

[Closure]

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

[Send]

$$\frac{\begin{array}{c}\forall i \in \{1 \ldots n\}.\ \ \kappa \vdash \mathtt{E_i} \Rightarrow C_i \\ \forall\ \kappa_j \in MCS(\kappa, l, \text{Msg}, \langle [\![\mathtt{E_1}]\!]_\kappa \ldots [\![\mathtt{E_n}]\!]_\kappa\rangle).\ \kappa_j \vdash Body(Proc(\kappa_j)) \Rightarrow C_j\end{array}}{\kappa \vdash \mathtt{send}_l\ \mathtt{Msg(E_1 \ldots E_h)} \Rightarrow \bigwedge_i C_i \wedge \bigwedge_{(\kappa_j,\, i)} \left(\begin{array}{c} Formal(i, \kappa_j) \supseteq_p^{f^{(i,\kappa_j)}} [\![\mathtt{E_i}]\!]_\kappa \wedge C_j \wedge \\ [\![\mathtt{send}_l\ \mathtt{Msg(E_1 \ldots E_n)}]\!]_\kappa \supseteq_p Var(\mathtt{result}, \kappa_j)\end{array}\right)}$$
$$\text{where } f^{(i,\kappa_j)} = Conformers(FormalDecl(i, Proc(\kappa_j))) \cap Subclasses(FormalDecl(i, Proc(\kappa_j)))$$

[Apply]

$$\frac{\begin{array}{c}\kappa \vdash \mathtt{E}_0 \Rightarrow C_0 \\ \forall i \in \{1 \ldots n\}.\ \ \kappa \vdash \mathtt{E_i} \Rightarrow C_i \\ \forall\ \kappa_j \in ACS(\kappa, l, [\![\mathrm{E}_0]\!], \langle [\![\mathtt{E_1}]\!]_\kappa \ldots [\![\mathtt{E_h}]\!]_\kappa\rangle\ .\ \kappa_j \vdash Body(Proc(\kappa_j)) \Rightarrow C_j\end{array}}{\kappa \vdash \mathtt{apply}_l\ \mathtt{E}_0\mathtt{(E_1 \ldots E_h)} \Rightarrow C_0 \wedge \bigwedge_i C_i \wedge \bigwedge_{(\kappa_j,\, i)} \left(\begin{array}{c} Formal(i, \kappa_j) \supseteq_p^{f^{(i,\kappa_j)}} [\![\mathtt{E_i}]\!]_\kappa \wedge C_j \wedge \\ [\![\mathtt{apply}_l\ \mathtt{E}_0\mathtt{(E_1 \ldots E_h)}]\!]_\kappa \supseteq_p Var(\mathtt{result}, \kappa_j)\end{array}\right)}$$
$$\text{where } f^{(i,\kappa_j)} = Conformers(FormalDecl(i, Proc(\kappa_j))) \cap Subclasses(FormalDecl(i, Proc(\kappa_j)))$$

[Body]

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

Fig. 9. Specification of general algorithm.

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 ($[\![\mathtt{V}]\!]_\kappa$ or $[\![\mathtt{B.F}]\!]_\kappa$). 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 ($[\![$`new C`$]\!]_\kappa$). The [Closure] rule is similar to the [New] rule, but it uses the environment contour key selection function to compute a set of sequences 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:

(1) The argument expressions to the send must be analyzed, and their constraints included in the constraints generated by the send ($\wedge_i C_i$).

(2) For each callee procedure contour (κ_j), three kinds of constraints are generated:

—actuals are assigned to formals: $\mathit{Formal}(i, \kappa_j) \supseteq_p^{f^{(i,\kappa_j)}} [\![\mathtt{E_i}]\!]_\kappa$,

—the callee's body is analyzed: C_j,

—and a result is returned: $[\![\mathtt{send}_l\ \mathtt{Msg}\ \mathtt{(E_1 \ldots E_n)}]\!]_\kappa \supseteq_p \mathit{Var}(\mathtt{result}, \kappa_j)$.

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 are 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 to satisfy the constraint that $\mathit{CallSites}(\kappa)(\mathtt{send}_l\ \mathtt{Msg}\,(\mathtt{E_1 \ldots E_n})) \supseteq \mathit{MCS}(\kappa, l, \mathrm{Msg}, \langle [\![\mathtt{E_1}]\!]_\kappa \ldots [\![\mathtt{E_n}]\!]_\kappa \rangle)$. Similarly the intercontour flow of data via instance variable loads and stores is represented by the constraints $\mathit{LoadSites}(\kappa)(\mathtt{B.\,F}) \supseteq \mathit{IVCS}(\mathtt{F}, [\![\mathtt{B}]\!]_\kappa)$ and $\mathit{StoreSites}(\kappa)(\mathtt{B.\,F}) \supseteq \mathit{IVCS}(\mathtt{F}, [\![\mathtt{B}]\!]_\kappa)$.

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 ($\mathtt{E_0}$) and the apply contour selector (ACS) is invoked to compute the set of callee contours.

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

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

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

The first rule states that if the identifiers of two instance variable contours are related, which implies that they represent 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. The second rule imposes a similar constraint on the class set map component of procedure contours. These constraints ensure that different degrees of context sensitivity can coexist, by requiring that when a class contour is stored in a set at one level of context sensitivity, then it (or some more conservative class contour) appears in the corresponding set of all more conservative views of the same source program construct. All of the algorithms described in subsequent sections trivially satisfy the second rule, and only k-l-CFA for $l > 0$ requires effort to satisfy the first.

4.5 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 as class contour sets grow and new procedures and/or procedure contours become reachable at call sites. The desired final solution is the greatest (most optimistic) call graph that satisfies the constraints. A number of algorithms are known for solving systems of set constraints [Aiken 1994]. Section 5 discusses the constraint satisfaction mechanisms used by the Vortex implementation framework.

The initial values assigned to the sets can also have a large impact on both the time required to compute the solution and the quality (precision) of the solution. An algorithm 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 values are 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 time consumed by constraint satisfaction without negatively impacting precision. Another possibility is to selectively give pessimistic initial values in the hope 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 noniterative pessimistic algorithms can be modeled by initializing all class contour sets to bottom.

5. VORTEX IMPLEMENTATION FRAMEWORK

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 are described in several previous papers [Grove 1995; Grove et al. 1997; DeFouw et al. 1998; Grove 1998].

5.1 Overview

The Vortex implementation of the general call graph construction algorithm closely follows the specification given in Section 4.4. It is divided into two main subsystems: constraint generation and constraint satisfaction.

The constraint generation subsystem is implemented directly from the specification in Figure 9, with extensions to support Cecil and Java language features. A method is defined on each kind of Vortex abstract syntax tree (AST) node 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.

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 5.3.1). 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 (intraprocedural) constraints and the propagation (as necessary) of the resulting class contour information along all of the contour's outgoing intercontour (interprocedural) dataflow edges, which may result in adding contours to the worklist.

5.2 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 generation and satisfaction 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 subsequent sections 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, we believe that this overhead affects the absolute cost of call graph construction only, not the relative cost of algorithms implemented in the framework or the asymptotic behavior of the algorithms.

5.3 Design Choices

5.3.1 *Implicit vs. Explicit Dataflow Graphs.* One of the most important considerations in the implementation of the framework is managing time/space tradeoffs. Most previous systems explicitly constructed 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 of algorithms to choose between 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 contour boundaries (those corresponding to formal parameters, local variables that are accessed by lexically nested functions, procedure return values, and instance variables) are actually persistently represented. All derived class sets and all intra- and interprocedural 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 5.4.2. Because reducing memory usage is very important, the Vortex implementations of algorithms that only generate inclusion constraints (0-CFA, k-l-CFA, CPA, and SCS) utilize the implicit representation.

5.3.2 *Iteration Order.* A key component of the constraint satisfaction subsystem is the worklist abstraction used to drive its iteration. Some possible implementations of a worklist include an implicit work stack via recursive functions, 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 worklist implementation can have a large impact on analysis time. A stack-based implementation yields a depth-first ordering of work, whereas a queue-based implementation yields a breadth-first ordering. Intuitively, a depth-first traversal 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 in the analysis of the caller. On the other hand, a breadth-first traversal has the advantage of potentially batching multiple reanalyses of a contour. For example, a contour is initially enqueued for reanalysis, since during the analysis of one of its callers it was determined that the argument class sets passed to the contour had 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 finally reaches the front of the queue. Informal performance tuning revealed that both of these effects are important and, as a result, the Vortex implementation uses a hybrid mostly-FIFO ordering. It uses a work queue, but the first time a contour is encountered, it is immediately analyzed via recursion rather than being enqueued for later analysis.

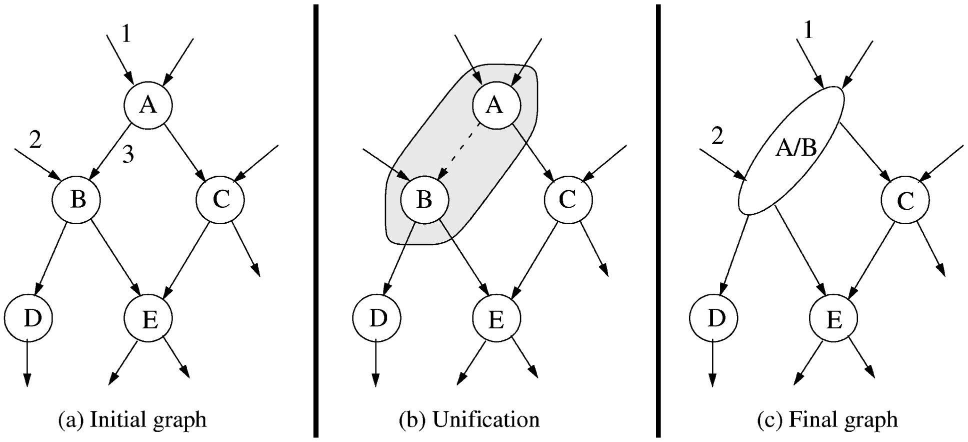
5.3.3 *Program Representation.* Interprocedural class analysis operates over a summarized AST representation of the program.[6] The summarized AST abstracts the program by collapsing all nonobject dataflows and by ignoring all intraprocedural control flow. Nonobject dataflow is the portion of a procedure's 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 (nonobject) integers would be represented only by a reference to the integer AST summary node. The second part of the summary 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 nonobject 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 intraprocedural 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 flow-insensitive AST-based analysis was roughly twice as fast.

5.4 Implementation of Key Primitives

5.4.1 *Bounded Inclusion Constraints.* Equality constraints have been used to define near-linear-time binding-time [Henglein 1991] and alias [Steensgaard 1996] 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

[6] This is an AST form of the compiler's language-independent internal repersentation, shared by Cecil and Java programs.



(a) Initial graph

(b) Unification

(c) Final graph

Fig. 10. Unification.

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 subsystem 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.

5.4.2 *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 subsystem is not allowed to incur any propagation costs on the behalf of effective equality constraints, an alternative satisfaction method must be used. Hence, once two nodes in the dataflow graph are linked by effective equality constraints, they are unified into a single node using fast union-find data structures [Tarjan 1975]. 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 with no additional work, since the two logical sets are now represented by the same actual set.

Figure 10 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 10(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 10(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`,

[7] The edge may have a filter which actually blocks the class contour from begin propagated across the edge. Regardless of whether or not the class passes the filter, the edge's counter decremented in order to maintain the guarantee of p-bounded work.

without incurring the cost of crossing the now nonexistent 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.

5.4.3 *Lazy Constraint Generation and Unification.* As discussed in Section 2, in an optimistic analysis the circular dependency among receiver class sets, the program call graph, and interprocedural analysis is resolved via iteration to a fixed-point. In this process, constraints are generated lazily as procedures are proven to be reachable, and thus 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 currently represents the set of unified nodes that includes B, and ensures 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 are filters on at least some edges in the graph, our forcible unification approach is overly pessimistic.

6. EXPERIMENTAL METHODOLOGY

This section describes our experimental methodology. The data presented in Sections 7 through 10 is a selected subset of that found in Grove's Ph.D. thesis [Grove 1998]. This article focuses on answering two primary questions:

—What are the costs of constructing a program's call graph using a particular call graph construction algorithm?

—To what extent do differences in call graph precision impact the effectiveness of client interprocedural analyses?

6.1 Benchmark Description

Experiments were performed on the Cecil [Chambers 1993] and Java [Gosling et al. 1996] programs described in Table I. With the exception of two of the Cecil programs, all of the applications are substantial in size. Before these experiments, experimental assessments of call graph construction algorithms have almost exclusively used benchmark programs consisting of only tens or hundreds of lines of source code; only a few of the previous studies included larger programs that ranged in size up to a few thousand lines of code. Program size is an important issue, since many call graph construction algorithms have worst-case running times that are polynomial in the size of the program. The data presented in Sections 8 and 9 demonstrate that these bounds are not just of theoretical concern; in practice, many of the algorithms exhibit running times

Table I. 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

[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 libray. All Java programs include a 16,400-line standard library.

[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 nonlibrary code and were developed by different people.

that appear to be super-linear in program size, and thus cannot be practically 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.

6.2 Experimental Setup

All experiments were conducted using the Vortex compiler infrastructure [Dean et al. 1996; Chambers et al. 1996]. The basic methodology was 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 [Hölzle and Ungar 1994; Grove et al. 1995] was demonstrated to be an important optimization for some object-oriented languages, but may not be desirable to include in every optimizing compiler, all of the experiments used 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, including intraprocedural class analysis [Johnson et al. 1988; Chambers and Ungar 1990]; hard-wired class prediction for common messages (Cecil only) [Deutsch and Schiffman 1984; Chambers and Ungar 1989]; splitting [Chambers and Ungar 1990]; class hierarchy analysis [Fernandez 1995; Dean et al. 1995; Diwan et al. 1996]; inlining, static class prediction [Dean 1996]; closure optimizations that identify and stack-allocate LIFO-closures and sink partially-dead closure creations (Cecil 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 and Ungar 1994; Grove et al. 1995]. For all programs with nontrivial 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 *A*, the **base+IP$_\mathbf{A}$** configuration augments **base** with the interprocedural analyses listed below, which enable the intraprocedural optimizations included in **base** to work better: more details on the individual interprocedural analyses, and experiments assessing their effectiveness, can be found in Grove [1998].

  —*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. Intraprocedural analyses can exploit this information to more accurately estimate the potential effect of noninlined calls on local dataflow information.

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

  —*Escape analysis*: Interprocedural escape analysis identifies first-class functions guaranteed not to out-live their lexically enclosing environment, thus enabling the function objects and their environments to be stack-allocated [Kranz 1988]. 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 benchmarks.

  —*Treeshaking*: As a side-effect of constructing the call graph, the compiler identifies those procedures 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$_\mathbf{A}$+pgcp** configuration augments the **base+IP$_\mathbf{A}$** 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$_\mathbf{A}$+pgcp** configurations, since any additional inlining enabled by interprocedural analysis would result in longer inlined chains of methods.

Thus, potentially more precise call-chain[8] profile information could be obtained by iteratively profiling the **base**+$\mathbf{IP_A}$+**pgcp** configuration. However, this effect should be quite small, and by using the same profile data for every configuration of a program, one 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 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.

6.3 Metrics

In this article we focus on the direct costs of call graph construction and the bottom-line impact of call graph precision on the effectiveness of interprocedural optimizations. Previous work [Grove 1998] contained much more extensive experimental results, including several metrics for call graph precision, and for secondary effects like the impact of call graph precision on interprocedural analysis time and the relative contributions of each of the five interprocedural analyses to improvements in application performance.

The primary cost of a call graph construction algorithm is the compile time expended running the algorithm. Hence, the primary metric to assess the direct cost of a particular algorithm is the CPU time[9] 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, since it does not account for details of the garbage collection algorithm, and thus may overestimate an algorithm's peak memory requirements.

Call graph precision can affect the quality of the information computed by the interprocedural analyses, thus affecting bottom-line application performance. To assess the importance of call graph precision, we focus on changes in application runtime (CPU time). Note that minor variations in application runtime may *not* be significant because instruction caching effects can cause application runtime to vary noticeably, independent of any optimizations enabled by interprocedural analysis. For example, simply running the `strip`[10] utility on an executable was observed to yield changes of up to +/−8% in application runtime.

[8] More information on call chain porfiles, their interactions with inlining, and the details of profile-guided receiver class prediction in Vortex can be found elsewhere [Grove et al. 1995].

[9] Combined system and user mode CPU time as reported by `rusage`.

[10] `strip` simply removes the symbol table and relocation bits from the executable.

7. LOWER AND UPPER BOUNDS ON THE IMPACT OF CALL GRAPH PRECISION

This section establishes lower and upper bounds for the potential importance of call graph precision on the bottom-line impact of the five interprocedural analyses for 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 provides context for the experimental assessment of the other call graph construction algorithms found in Sections 8 through 10.

7.1 Algorithm Descriptions

7.1.1 *"Lower Bound" Algorithms.* A true lower bound on the benefit of call graph precision could be obtained by interprocedural analysis using $G_\bot$, the call graph in which every call site is assumed to invoke every procedure in the program. However, $G_\bot$ is needlessly pessimistic; a much more precise call graph, G_{selector}, can be constructed with only superficial analysis of the program. G_{selector} improves $G_\bot$ 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[11] to disambiguate any static overloading of message names, G_{selector} also takes 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 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 a message within the body of a method 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 hierarchies, exploiting specialized formal parameters is just a special case of exploiting static type declarations; but in languages like Cecil, the two sources of information can be different. A number of systems have used class hierarchy analysis to resolve message sends and build program call graphs [Fernandez 1995; Dean et al. 1995; Diwan et al. 1996; Bacon and Sweeney 1996; Bairagi et al. 1997]. 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 ($G_{\text{reachable}}$). 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 class hierarchy analysis and optimistic reachability analysis to build a call graph [Bacon and Sweeney 1996].

Finally, as in Diwan's Modula-3 optimizer [Diwan et al. 1996], intraprocedural class analysis could be used to increase the precision of G_{selector}, G_{CHA},

[11] 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.

$G_{\text{reachable}}$, or G_{RTA} by improving the analysis of any messages sent to objects created within the method.

7.1.2 *"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. If the IP-G_{prof_i} optimized program was run on any other input, it could compute an incorrect result.

In theory, G_{prof_i} is at least as precise as G_{ideal}. However, limitations in Vortex's profiling infrastructure[12] 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 ideal context-insensitive call graph. In the other direction, G_{prof_i} may also be too loose an upper bound if the single run does not fully exercise the program. We believe that this second concern (too loose an upper bound) is unlikely to be a significant problem for these benchmark programs. The input data sets we 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, we built 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. Optimization based on this combined G_{prof} resulted in only a 2% slowdown relative to the single-run G_{prof}.

7.2 Experimental Assessment

To establish lower and upper bounds on the bottom-line impact of call graph precision, interprocedural analysis was done using two of the call graph construction algorithms discussed above: G_{selector} and G_{prof}. Figure 11 displays normalized execution speeds for the **base**, **base+IP$_{\text{selector}}$**, **base+IP$_{\text{prof}}$**, **base+pgcp**, **base+IP$_{\text{selector}}$+pgcp**, and **base+IP$_{\text{prof}}$+pgcp** configurations of each application. Throughout this article we use the convention of stacking the bars for the two configurations that differ only in the presence or absence of profile-guided class prediction, with the static bar shaded and the additional speedups enabled by profile-guided class prediction, shown by the white bar.

[12] Vortex's profiling infrastructure is described in more detail elsewhere [Grove et al. 1995]. The key issue is that the profile-derived class distributions grathered 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.

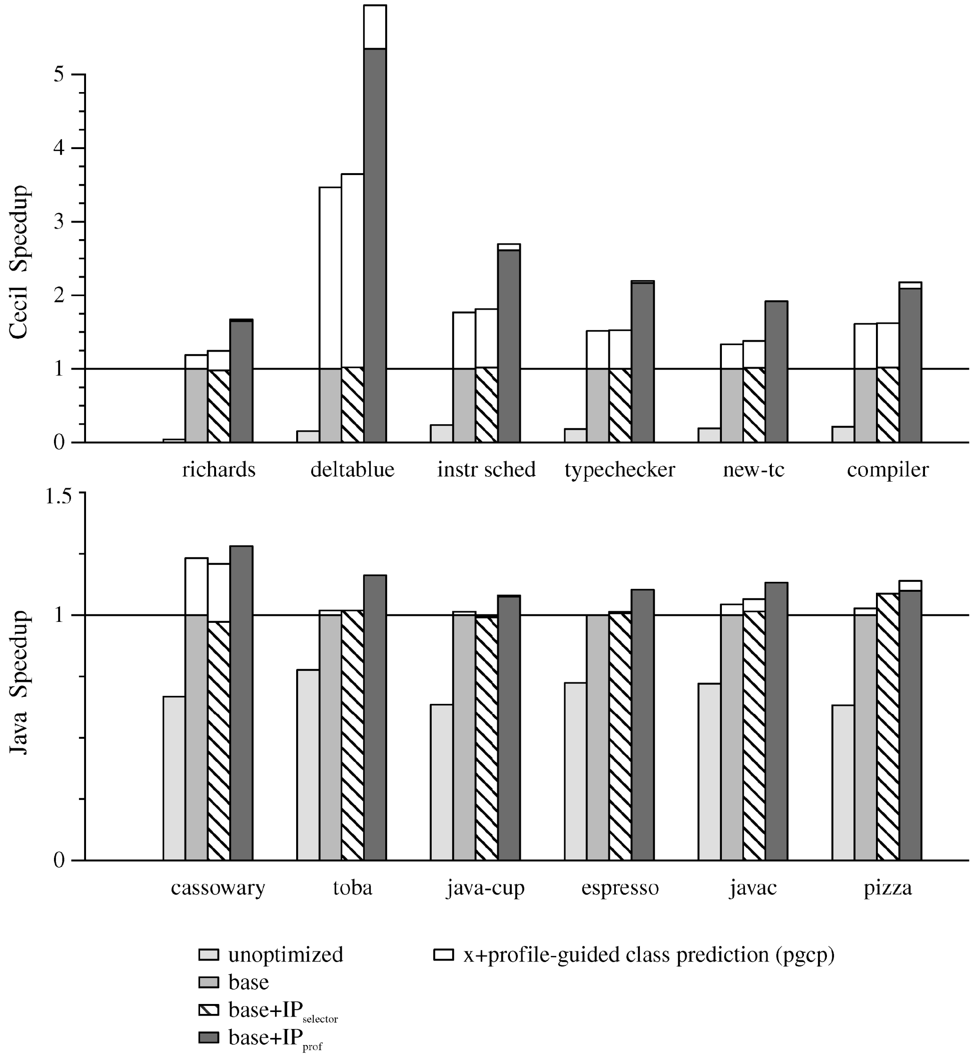


Fig. 11. Upper and lower bounds on the impact of call graph precision.

The absence of the white bar indicates that profile-guided class prediction had no impact on performance; the case in which x+pgcp is measurable slower than x is explicitly noted in the text. To emphasize that **base** is already highly optimized, an **unopt** configuration in which no Vortex optimizations were performed (but the resulting C files were still compiled by gcc -O2) is shown as well; the difference between the **unopt** and **base** configurations indicates the effectiveness of Vortex's basic optimization suite. The difference between the **base** and **IP$_{\mathbf{prof}}$** configurations approximates the maximum speedup achievable in the current Vortex system by an arbitrarily precise call graph construction algorithm. The **IP$_{\mathbf{selector}}$** 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 $\mathbf{IP_{selector}}$ and $\mathbf{IP_{prof}}$ configurations.

For almost all the programs, the performance difference between $\mathbf{base{+}IP_{selector}}$ and $\mathbf{base{+}IP_{prof}}$ was almost as large as that between **base** and $\mathbf{base{+}IP_{prof}}$, indicating that more aggressive call graph algorithms than $G_{selector}$ are required to reap the potential benefits of interprocedural analysis. For the Cecil programs, the potential benefit is quite large. $\mathbf{base{+}IP_{prof}}$ dominates **base+pgcp**, and on the four larger benchmarks yields speedups ranging from a factor of 1.9 to 2.6 over **base**. The potential improvements for the Java benchmarks are much smaller; this is not entirely unexpected, since the combination of Java's hybrid object model and its static type system both make **unopt** more efficient than it is in Cecil, and make the optimizations in the **base** configuration more effective, thus leaving even less overhead for interprocedural analysis to attack. For cassowary, $\mathbf{base{+}IP_{prof}}$ is roughly 25% faster than **base**, and the potential benefits are smaller for the remainder of the Java programs.

8. THE BASIC ALGORITHM: 0-CFA

The 0-CFA algorithm is the classic context-insensitive, dataflow-sensitive call graph construction algorithm. It produces a more precise call graph than $G_{selector}$, and the other "lower bound" algorithms of Section 7.1.1, 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 [1988, 1991]. The basic 0-CFA strategy has been used in a number of call graph construction algorithms; Palsberg and Schwartzbach's basic, algorithm [1991]; Hall and Kennedy's [1992], call graph construction algorithm for Fortran; and Lakhotia's [1993] algorithm for building a call graph in languages with higher-order functions are all derived from 0-CFA.

8.1 Algorithm Description

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

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

For every procedure, instance variable, and class, the 0-CFA contour key selection functions return the singleton set $\{\perp\}$, resulting in the selection of the

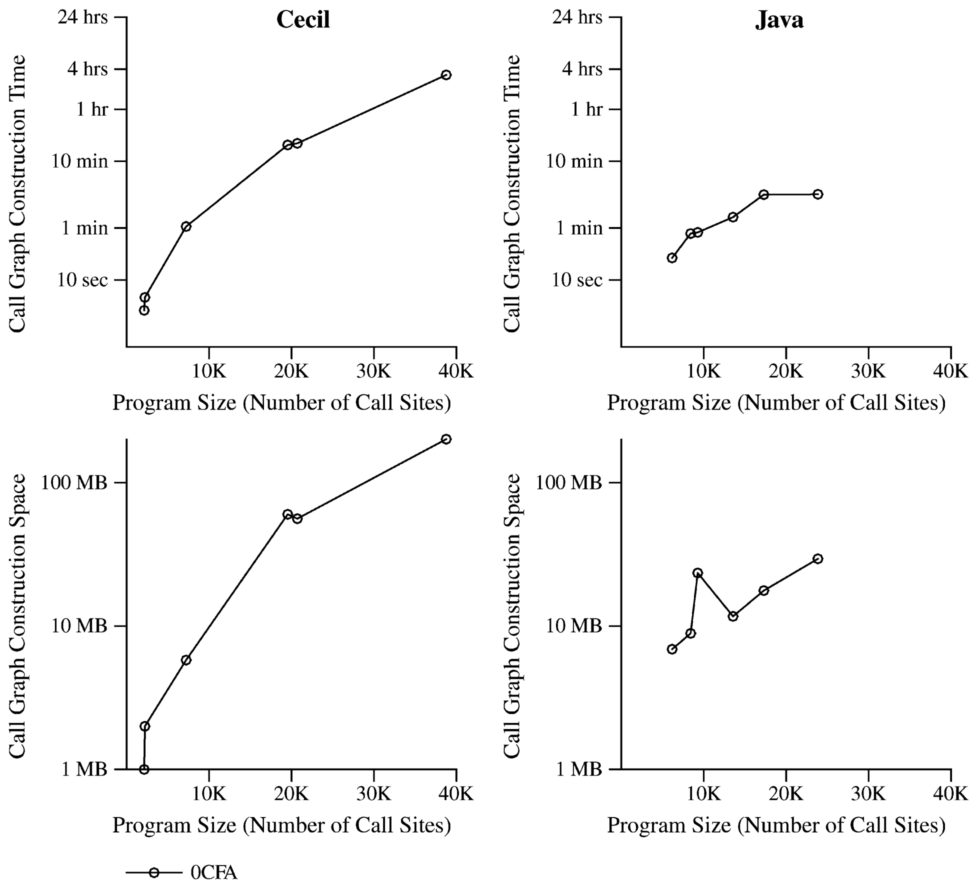


Fig. 12. Costs of 0-CFA call graph construction.

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 nonspecified class contour sets are optimistically initialized to the empty set.

8.2 Experimental Assessment

Figure 12 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-axes 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 were 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) are 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. We 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 so 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 javac contains only 297 classes and no closures.

Figure 13 shows the bottom-line performance impact of 0-CFA by displaying the execution speeds of the two $\mathbf{IP_{0\text{-}CFA}}$ configurations of each benchmark (the third pair of bars). The execution speeds obtained by the **base**, $\mathbf{IP_{selector}}$, and $\mathbf{IP_{prof}}$ configurations are repeated to enable easy comparison. For the two smallest Cecil programs, 0-CFA enables almost all of the potentially available speedup embodied in the $\mathbf{IP_{prof}}$ configuration. In the four larger Cecil programs, 0-CFA enables roughly half of the potentially available speedup: it substantially outperforms **base**, but the remaining large gap between it and $\mathbf{IP_{prof}}$ indicates that context sensitivity might enable additional performance gains. The critical difference between the small and large Cecil programs 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 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.

9. CONTEXT-SENSITIVE ALGORITHMS

In contrast to the 0-CFA algorithm, context-sensitive algorithms 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 these algorithms generate only simple inclusion constraints, $\supseteq_\infty$.

9.1 Algorithm Descriptions

9.1.1 *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 [1981]). If $k = 0$, then this degenerates to the single-point lattice and a context insensitive algorithm (0-CFA);

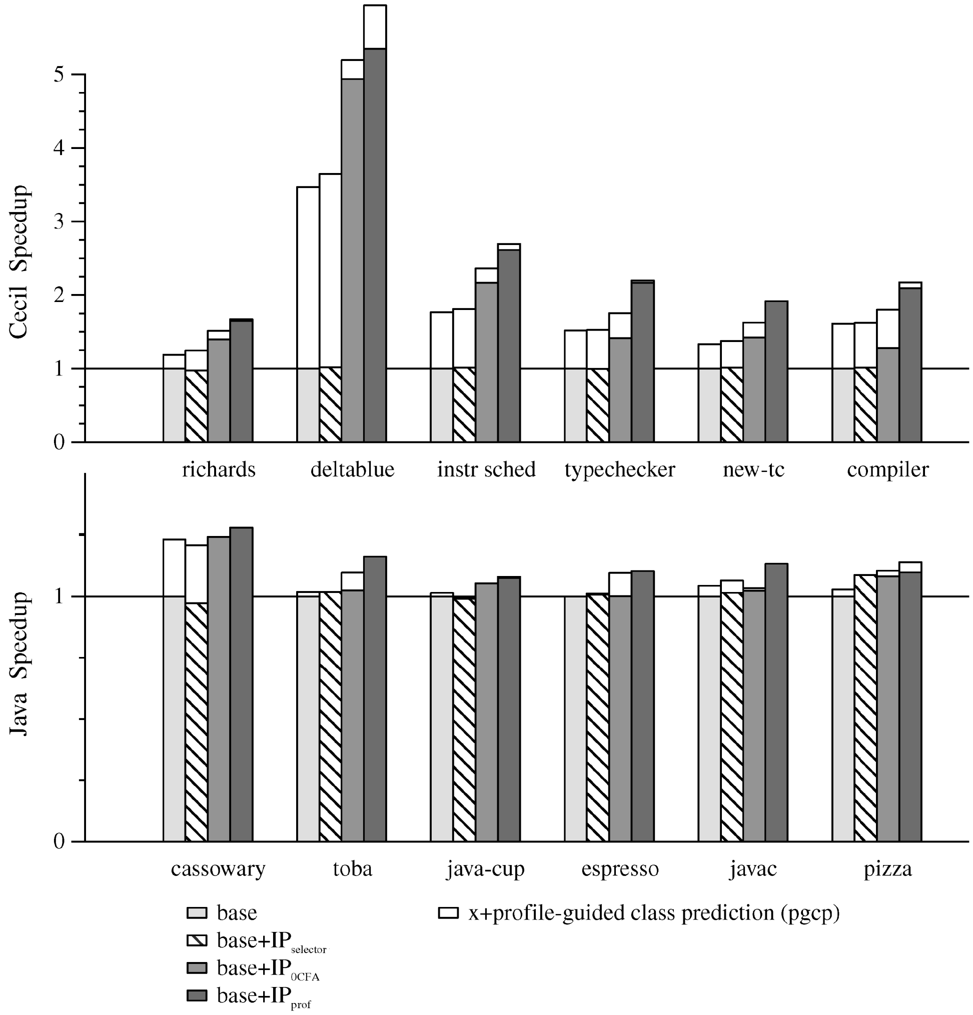


Fig. 13. Performance impact of 0-CFA.

$k = 1$ 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 1988; Shivers 1991]; Oxhøj's 1-CFA extension to Palsberg and Schwartzbach's algorithm [Oxhøj et al. 1992], or various other adaptations of k-CFA to object-oriented programs [Vitek et al. 1992; Phillips and Shepard 1994]. More sophisticated adaptive algorithms try to use different levels of k in different regions of the call graph to more flexibly manage the tradeoff between analysis time and precision [Plevyak and Chien 1994; Plevyak 1996]. 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 1979]; Callahan's extension to Ryder's work to support recursion [Callahan et al. 1990]; and Emami's alias analysis algorithm for C [Emami et al. 1994] all treat each acyclic path through the call graph as creating a new context. Alt and Martin developed an even more aggressive call graph construction algorithm, used in their PAG system, that first "unrolls" levels of recursion [Alt and Martin 1995]. 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 1994].

For object-oriented programs, Shivers's k-CFA family of algorithms can be extended straightforwardly 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. 1992; Phillips and Shepard 1994]. The partial orders used as contour keys by these algorithms are detailed below:

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

The contour key selection functions are defined using two auxiliary functions $\oplus$ 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:

$$PKS(caller : \text{ProcContour}, c : CS, a : \text{AT(CCS)}, callee : P)$$
$$\rightarrow \{F_k(Proc(ID(caller)), Key(ID(caller))))\}$$
$$IVKS(i : \text{InstVariable}, b : \text{ClassContourSet}) \rightarrow \left(\begin{array}{l} \{\bot\}, l = 0 \\ b, \text{otherwise} \end{array} \right.$$
$$CKS(c : \text{Class}, n : \text{NewSite}, p : \text{ProcContour})$$
$$\rightarrow \left(\begin{array}{l} \{\bot\}, l = 0 \\ \{\langle F_1(Proc(ID(p)), Key(ID(p))))\rangle\}, \text{otherwise} \end{array} \right.$$
$$EKS(c : \text{Closure}, p : \text{ProcContour})$$
$$\rightarrow \left(\begin{array}{l} \{\bot\}, \text{if } c \text{ contains no non-global free variables} \\ \{Key(ID(p)) \oplus lc \mid lc \in Lex(ID(p))\}, \text{otherwise} \end{array} \right.$$
$$CIF : p = \infty$$
$$SIF : \emptyset$$

Thus, the procedure context-sensitivity strategy used in the k-l-CFA family of algorithms is identical to that 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 $l \leq k + 1$, since *ClassKeys* are based on *ProcKeys*, and larger values of l do not have any additional benefit.

9.1.2 *Context Sensitivity Based on Parameters.* Another common basis for context-sensitive analysis of procedures is some abstraction of the actual parameter values passed to the procedure from its call sites. For example, an abstraction of the alias relationships among actual parameters was used as the basis for context sensitivity in algorithms for interprocedural alias analysis [Landi and Ryder 1991; Wilson and Lam 1995]. 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 so 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 1995] and the Simple Class Set algorithm (SCS) [Grove et al. 1997]. 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, both generate inclusion constraints ($p = \infty$), and both initialize class contour sets to the empty set. The two algorithms share the following common parameters:[13]

$$
\begin{aligned}
&\mathit{IVKS}(i : \text{InstVariable}, b : \text{ClassContourSet}) \rightarrow \{\bot\}\\
&\mathit{CKS}(c : \text{Class}, n : \text{NewSite}, p : \text{ProcContour}) \rightarrow \{\bot\}\\
&\mathit{EKS}(c : \text{Closure}, p : \text{ProcContour})\\
&\quad\rightarrow \left(\begin{array}{l}\{\bot\}, \text{if } c \text{ contains no non-global free variables}\\ \{\mathit{Key}(\mathit{ID}(p)) \oplus\ lc \mid lc \in \mathit{Lex}(\mathit{ID}(p))\}, \text{otherwise}\end{array}\right.\\
&\mathit{CIF} : p = \infty\\
&\mathit{SIF} : \emptyset
\end{aligned}
$$

For its procedure contour key selection function, CPA uses

$$
\begin{aligned}
&\mathit{PKS}(\mathit{caller} : \text{ProcContour}, c : \mathit{CS}, a : \text{AllTuples(ClassContourSet)},\\
&\quad \mathit{callee} : \text{Proc}) \rightarrow \mathit{CPA}(a)\\
&\quad\ \text{where } \mathit{CPA}(\langle S_1, S_2, \ldots, S_n\rangle) = S_1 \times S_2 \times \ldots \times S_n
\end{aligned}
$$

while SCS uses

$$
\begin{aligned}
&\mathit{PKS}(\mathit{caller} : \text{ProcContour}, c : \mathit{CS}, a : \text{AllTuples(ClassContourSet)},\\
&\quad \mathit{callee} : \text{Proc}) \rightarrow \{a\}
\end{aligned}
$$

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

[13] Agesen's implementation of CPA for Self actually uses a more complex *EKS* function that 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 [1996] for more details.

callee procedure, the procedure contour key selection function returns a single contour key, which is exactly the tuple of actual parameter class contour sets a, thus only a single contour per callee procedure is selected for 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 generated by the first call site. In contrast, SCS can only reuse a callee contour if there are two call sites that have identical tuples of class contour sets passed as actual parameters. On the other hand, CPA may 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 14 contains a program and the resulting SCS and CPA call graphs. Each node 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. 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 result than SCS. In the CPA call graph, the contours representing `twice(@num)` only invoke contours for `+(@int,@int)` and `+(@float,@float)`. However in the SCS call graph, `twice(@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 is polynomial in the size of the program [Agesen 1996]. In the worst case, SCS may require $O(N^{N^a})$ contours to analyze a program. To avoid requiring an unreasonably large number of contours in practice, Agesen actually implements a variant of CPA, which 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.[14] 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 9.2 only present data for the b-CPA and SCS algorithms. Unbounded CPA does not scale to large Cecil programs [Grove et al. 1997]. 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, since 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 occurs (thus enabling a widening operation to be applied in the procedure

[14] Our implementation of b-CPA uses a threshold value of 10.

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

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

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

```
procedure A() {
   if (random()) {
      x := 3; y := 5;
   } else {
      x := 3.5; y := 4.5;
   }
   return x + double(y);
}
procedure B() {
   if (random()) {
      x := 3; y := 5;
   } else {
      x := 3.5; y := 5;
   }
   return x + double(y);
}
```

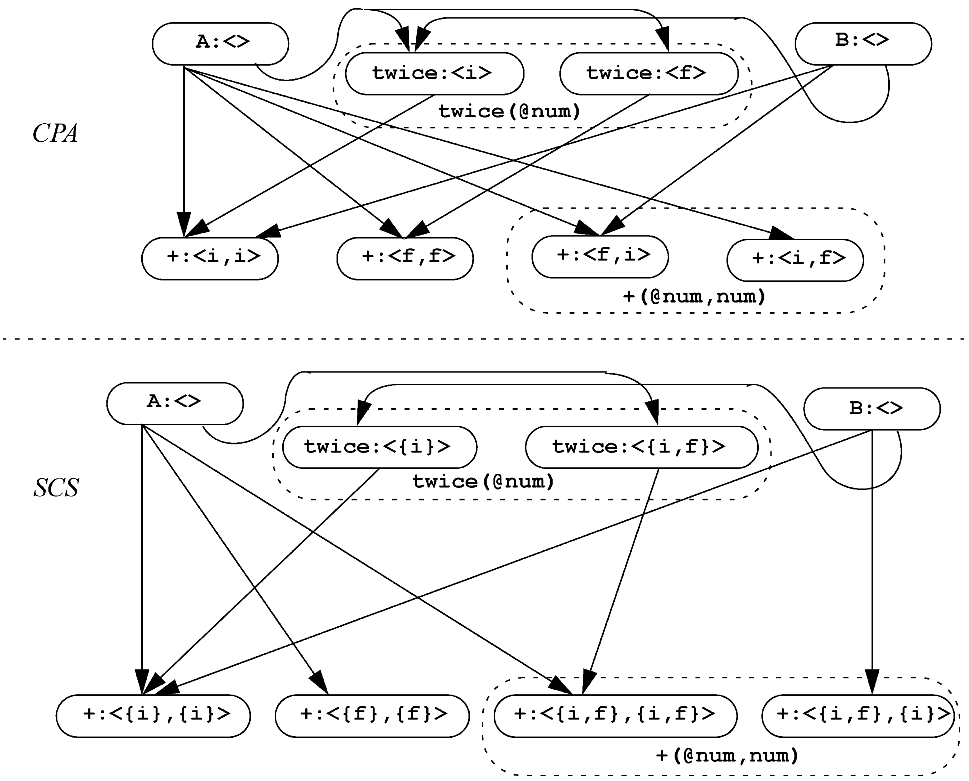


Fig. 14. CPA and SCS example.

contour key selection function) [Agesen 1996]. 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 were defined by exploiting similar ideas. The "eager splitting" used as a component of each phase of Plevyak's iterative refinement algorithm [Plevyak 1996] is equivalent to unbounded CPA; thus an implementation of Plevyak's algorithm would also not scale to large Cecil programs. Pande's algorithm for interprocedural class analysis in C++ [Pande and Ryder 1994] is built on Landi's alias analysis for C [Landi and Ryder 1991] 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. 1993]. 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 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 1996].

9.2 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 15; 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 24 hours of CPU time. In particular, none of the context-sensitive algorithms could successfully analyze compiler—only 1-0-CFA completed on typechecker and new-tc, and higher values of *k*-*l*-CFA failed to complete on the three largest Java 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 16 shows the bottom-line benefits of the additional call graph precision enabled by context-sensitive interprocedural class analysis; for comparison, the speedups obtained by $\mathbf{IP}_{\mathbf{0\text{-}CFA}}$ and $\mathbf{IP}_{\mathbf{prof}}$ are also shown. The bars are grouped into three sets: the **base**, $\mathbf{IP}_{\mathbf{selector}}$, and $\mathbf{IP}_{\mathbf{0\text{-}CFA}}$ configurations, the $\mathbf{IP}_{\boldsymbol{k\text{-}l}\mathbf{\text{-}CFA}}$ configurations, and the $\mathbf{IP}_{\mathbf{b\text{-}CPA}}$, $\mathbf{IP}_{\mathbf{SCS}}$, and $\mathbf{IP}_{\mathbf{prof}}$ configurations. Missing bars indicate combinations that did not complete. Overall, despite some improvements in precision, none of the context-sensitive analyses 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 $\mathbf{IP}_{\mathbf{prof}}$ configurations, suggesting that further improvements may be possible if a scalable and more precise context-sensitive algorithm could be developed. Overall, at least for these

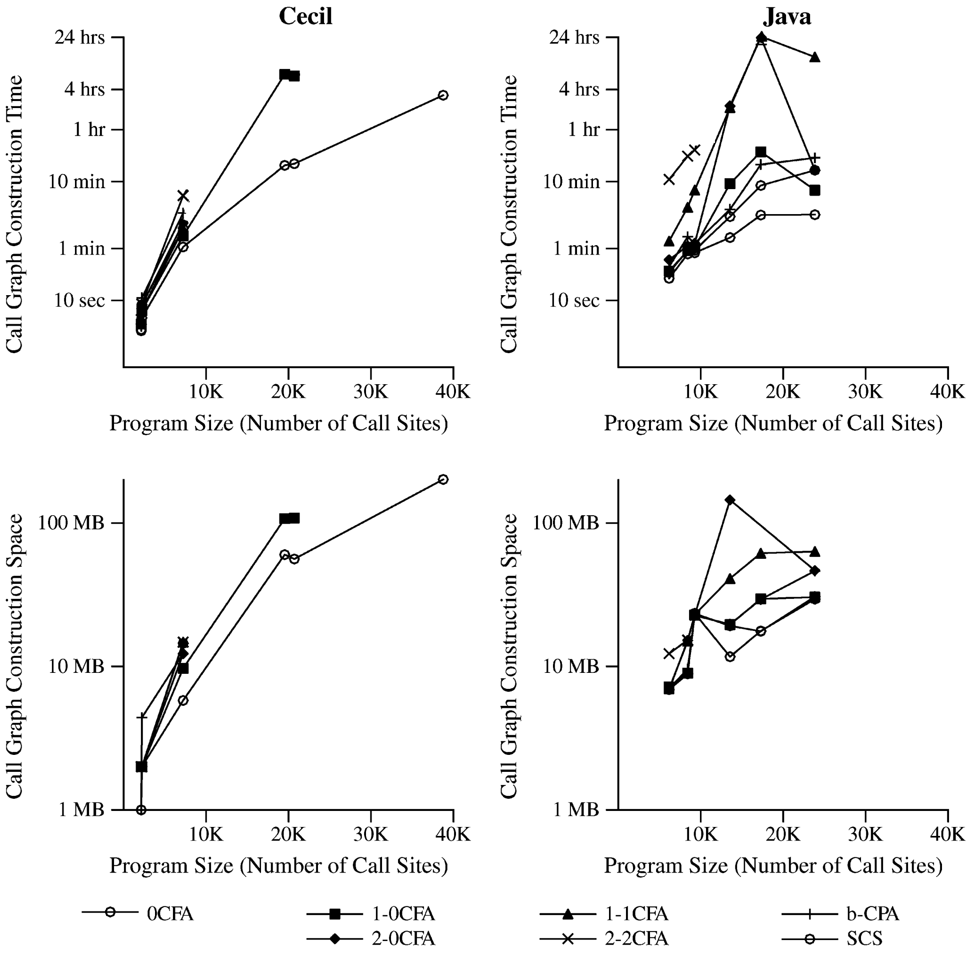


Fig. 15. Costs of context-sensitive call graph construction.

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.

10. 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 improvements. The algorithms 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.

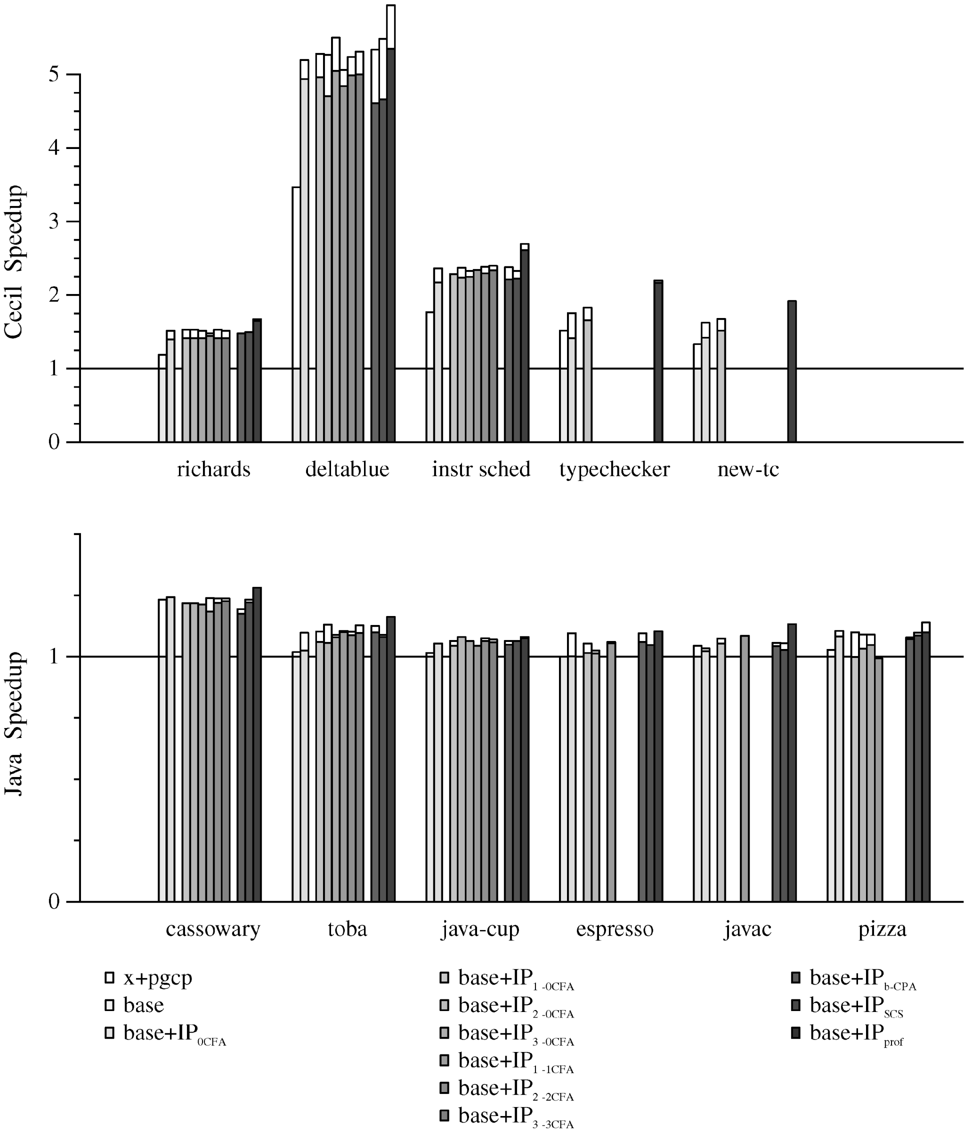


Fig. 16. Performance impact of context-sensitive algorithms.

10.1 Algorithm Descriptions

The 0-CFA algorithm only generates inclusion constraints. The basic constraint satisfaction method for inclusion constraints is to propagate class contour information from sources to sinks in the program dataflow graph until a fixed point is reached. This solution method results in a worst-case time complexity of $O(N^3)$, where N is a measure of program size. This follows from the fact that there are $O(N^2)$ 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 dataflow graph, that requires only O(N) time to compute a solution [Heintze and McAllester 1997]. However, it applies only to programs with bounded-size types, and may require O(N^2) time to answer a query about the results. In contrast to subtransitive 0-CFA, the approximation techniques defined in this section are applicable to any program, but yield less precise analyses than 0-CFA.

10.1.1 *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 10.1.6 use a uniform value of p throughout their constraint graph, but a number of other strategies are possible [Grove 1998]. As described previously (in Section 5.4.2), once p distinct class contours are propagated across a p-bounded inclusion constraint, the source and sink nodes connected by the constraint are unified, thus guaranteeing that the constraint is 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 dataflow-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 1991] and alias analysis [Steensgaard 1996]. Ashley explored reducing propagation costs by utilizing bounded inclusion constraints in a control flow analysis of Scheme programs [Ashley 1996, 1997]. In addition to approximating 0-CFA, he also applied the same technique to develop an approximate 1-CFA analysis. Shapiro and Horwitz have developed a family of alias analysis algorithms that mixes propagation and unification in a different way, 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 and Horwitz 1997]. But instead of using bounded inclusion constraints, Shapiro and Horwitz 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 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.

10.1.2 *Static Unification.* Recent algorithms proposed by Tip and Palsberg [2000] use static criteria to *a priori* unify nodes before the propagation phase begins. No nodes are dynamically unified during propagation, thus avoiding the complexity of implementing unification via fast union-find data structures [Tarjan 1975]. Their algorithms have the same O(N^3) worst-case time complexity as 0-CFA, but in practice may scale more effectively because the *a priori* unification can significantly reduce the magnitude of N. They propose and empirically evaluate four algorithms (static unification criteria):

—*CTA* maintains one distinct node for each class in the program. In effect, it unifies the nodes of all program constructs that have the same declaring/containing class.

—*MTA* maintains one distinct node for each class and each instance variable in the program. In effect, for each class it unifies the nodes of all variables declared in the class's methods into a single summary node for the class.

—*FTA* maintains one distinct node for each class and each method in the program. In effect, it unifies the nodes of all instance fields declared in the same class into a single node that represents all instance variables of the class. It also unifies the nodes of all variables declared in a method into a single node that summarizes the method.

—*XTA* maintains one distinct node for each instance field and method in the program. In effect, it unifies the nodes of all variables declared in a method into a single node that summarizes the method.

They experimentally evaluated their algorithms on a suite of Java programs and determined that XTA represented the best cost/precision tradeoff of the four. All four of their algorithms unify (at least) all of the nodes representing a method's variables. An important benefit of this approximation in the context of Java is that it allows them to avoid analyzing the method's bytecodes to discover its intraprocedural dataflow.

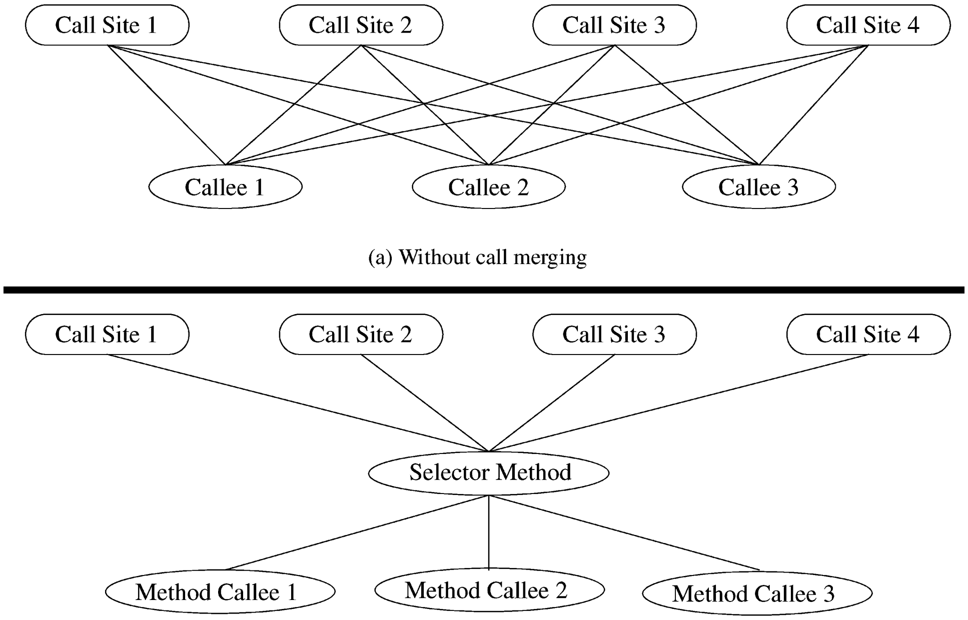
Declared-type analysis (DTA) combines additional pessimistic assumptions described below (Section 10.1.5) with static unification of all local variables that have the same declared type [Sundaresan et al. 2000].

10.1.3 *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.[15] Figure 17 illustrates call merging; Figure 17(a) shows the unmerged connections required to connect four call sites with the same selector to three callee procedures with that selector; Figure 17(b) shows the connections required if call merging is utilized. To simplify the picture, only a single edge is 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 so enables a reduction in worst-case analysis time of O(N) [DeFouw et al. 1998]. 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, it is 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 prepass that transforms the program source prior to running the call graph construction algorithm. The prepass

[15] 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.



(a) Without call merging

(b) With call merging

Fig. 17. Dataflow graph without and with call merging.

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" callees. All call sites of the original program are replaced by direct calls to the appropriate selector method.

10.1.4 *Closure Merging.* Closure merging reduces the number of classes in the program. 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. This approximation does not change the asymptotic worst-case complexity of an algorithm, but can still make a substantial impact on analysis time. For example, in typechecker, closure merging reduces the number of classes in the program from 4,535 to 655. Vortex implements both families of algorithms from Section 10.1.6 with and without closure merging; previous experiments have shown that closure merging reduces analysis time by roughly a factor of two, while in almost all cases having no measurable impact on bottom-line application performance.

10.1.5 *Pessimistic Assumptions.* As discussed in Section 2, the circular dependencies between receiver class sets, the program call graph, and interprocedural analysis could be broken by making a conservative assumption about one of the quantities and then computing the other two. In variable-type analysis (VTA) [Sundaresan et al. 2000], an initial call graph is constructed using Rapid

Type Analysis (RTA) [Bacon and Sweeney 1996]. Based on this call graph, 0-CFA style constraints between program variables are constructed, any strongly connected components in the resulting dataflow graph are collapsed, and class information is propagated in a single noniterative pass. Although collapsing the strongly connected components enables VTA to avoid iteration, it precludes using filters to exploit static type information. Sundaresan et al. empirically demonstrate that VTA is significantly more precise than RTA, but do not compare it to 0-CFA. Thus, it is unclear how much precision is lost or analysis time is gained by their approximations.

10.1.6 *Algorithms.* We present two families of algorithms that approximate 0-CFA using combinations of unification, call merging, and closure merging. Both families of algorithms use a simplistic strategy for generating bounded inclusion constraints, with a uniform constant value of p to control unification vs. propagation decisions. The two families are distinguished by the presence or absence of call merging: the first, p-Bounded, does not use call merging, whereas the second, p-Bounded Linear-Edge, does.

For constant values of p, instances of the p-Bounded algorithm have a worst-case time complexity of $O(N^2\alpha(N,N))$[16] 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. [1998] called this degenerate case of p-Bounded Linear-Edge, Linear-Edge 0-CFA.

The 0-Bounded Linear-Edge algorithm is very similar to Bacon and Sweeney's Rapid Type Analysis (RTA) [Bacon and Sweeney 1996]. 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. 1998].

10.2 Experimental Assessment

The experimental results, one pair of graphs per program, are found in Figure 18. Each graph plots two lines, one for p-Bounded and one for p-Bounded Linear-Edge, as p varies from 0 to ∞ (denoted N) along the x-axis. Analysis cost is reported by the first graph in each pair, which shows call graph construction time in CPU seconds. The second graph in each pair reports application execution speed (normalized to **base**). In the second graph, lines are plotted for both the **base+IP** and **base+IP+pgcp** configurations. Memory

[16] $\alpha(N,N)$ is the inverse Ackermann's function introduced by the fast union-find data structures. In practice, $\alpha(N,N) \leq 4$.

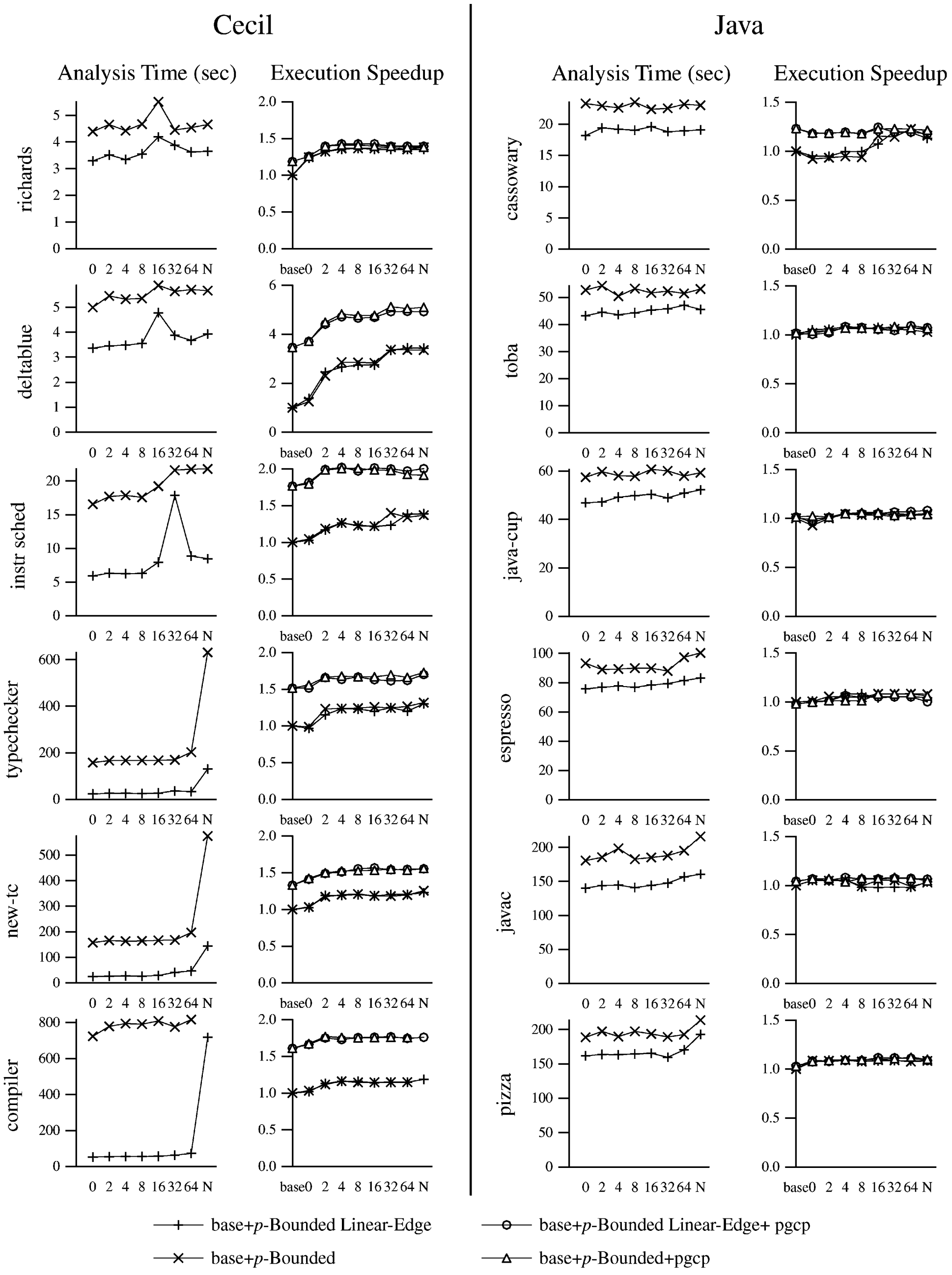


Fig. 18. Approximations of 0-CFA.

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 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 1 GB, 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. The excessive memory consumption of the ∞-Bounded algorithm indicates the importance of a careful choice between implicit and explicit representations of the program's dataflow graph (Section 5.3.1). The ∞–Bounded algorithm is semantically identical to 0-CFA, but in our implementation 0-CFA uses an implicit representation of the dataflow graph. Using a different representation enabled 0-CFA to analyze new-tc in 56 MB, typechecker in 107 MB, and compiler in 201 MB.

Surprisingly, across all the benchmarks, the additional precision of the near-quadratic-time p-Bounded algorithm does not enable any measurable performance improvement over that obtained by 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 call graph precision enabled by using bounded inclusion constraints instead of equality constraints ($p =$ a small constant vs. $p = 0$) translates into significant improvement in bottom-line application performance. Finally, for many programs, there is an interesting inverse-knee in the analysis time curves for intermediate values of p. This effect 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 dataflow graph does not become significantly smaller, but more propagation work can be incurred along edges between dataflow nodes before they are eventually unified.

11. SUMMARY COMPARISON OF CALL GRAPH CONSTRUCTION ALGORITHMS

Figure 19 depicts the relative precision of all the call graph construction algorithms described in this article (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

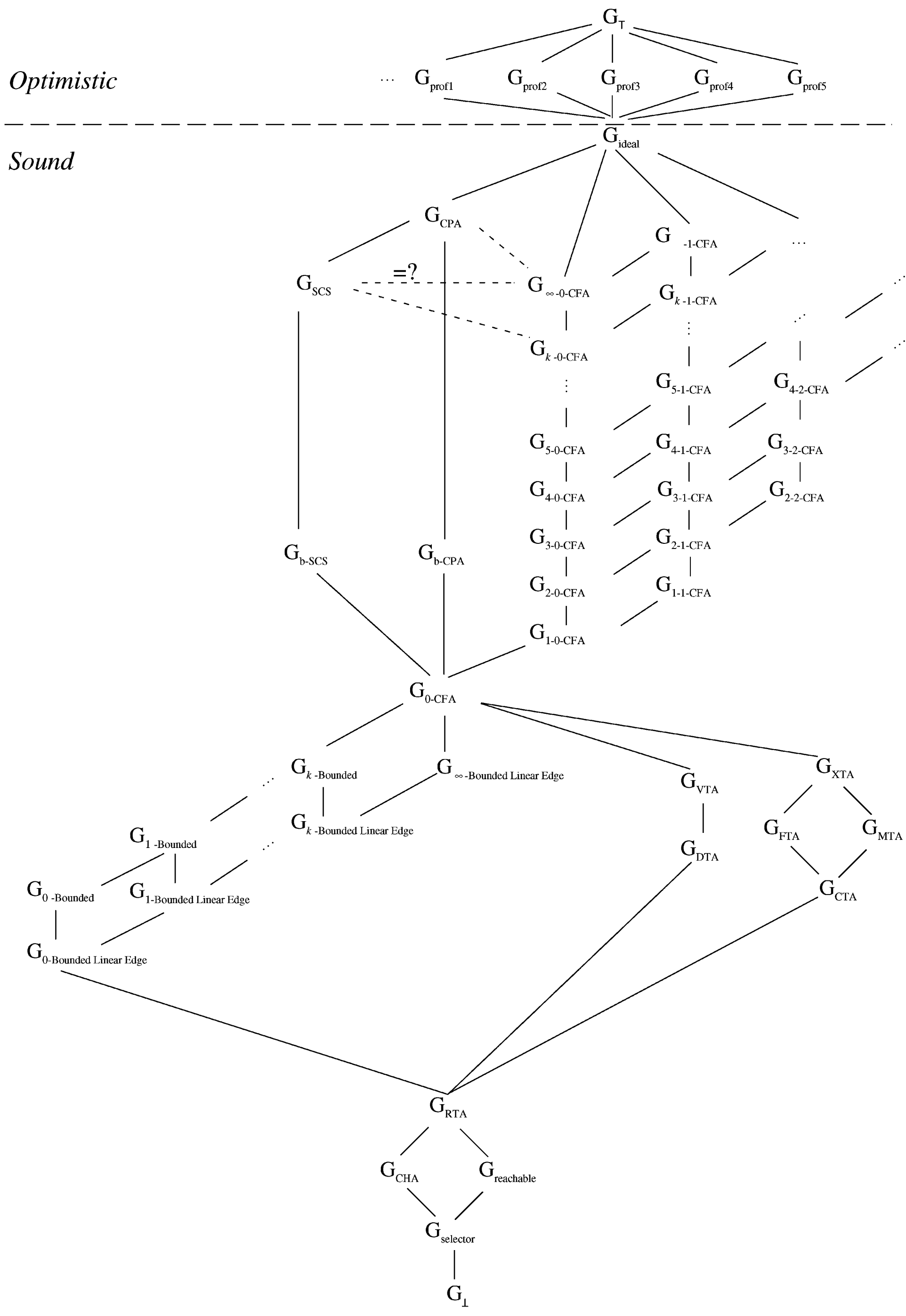


Fig. 19. Relative precision of computed call graphs.

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 sublattice. Increasing the degree of context 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 sublattices 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 (shown by dotted lines in Figure 19). All four of these algorithms use the same (context-insensitive) strategy to analyze data structures. Therefore, any precision differences are 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. For some programs, CPA and SCS produce call graphs that are more precise than k-0-CFA. More interestingly, there are programs for which CPA generates call graphs that are more precise than those produced by ∞-0-CFA. Figure 20(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 20(c) repeats the example from Section 9.1.2 that was used to illustrate how CPA can be more precise than 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 `twice`. The CPA call graph does not contain this inaccuracy. We conjecture that CPA is more precise than the other three algorithms and that SCS and ∞-0-CFA are equally precise.

To summarize the costs and benefits of using different call graph construction algorithms as the basis for interprocedural analysis and subsequent optimization, we repeat a subset of the experimental results. The $\mathrm{G}_{\mathrm{selector}}$ and $\mathrm{G}_{\mathrm{prof}}$

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

(c) Program two

Fig. 20. Example programs for call chain vs. parameter context sensitivity.

algorithms (Section 7) represent lower and upper bounds on the performance impact achievable in Vortex through interprocedural analysis. The 8-Bounded Linear-Edge (Section 10), 0-CFA (Section 8), and SCS (Section 9) 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 typically $2 < p < 16$.

Figure 21 shows the normalized execution speeds obtained by the **base+IP** and **base+IP+pgcp** configurations of the five call graph construction algorithms. The **base** and **base+pgcp** configurations are also shown for comparison. Figure 22 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 23 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 was plotted to show the cost/benefit of the algorithm relative to the cost/benefit

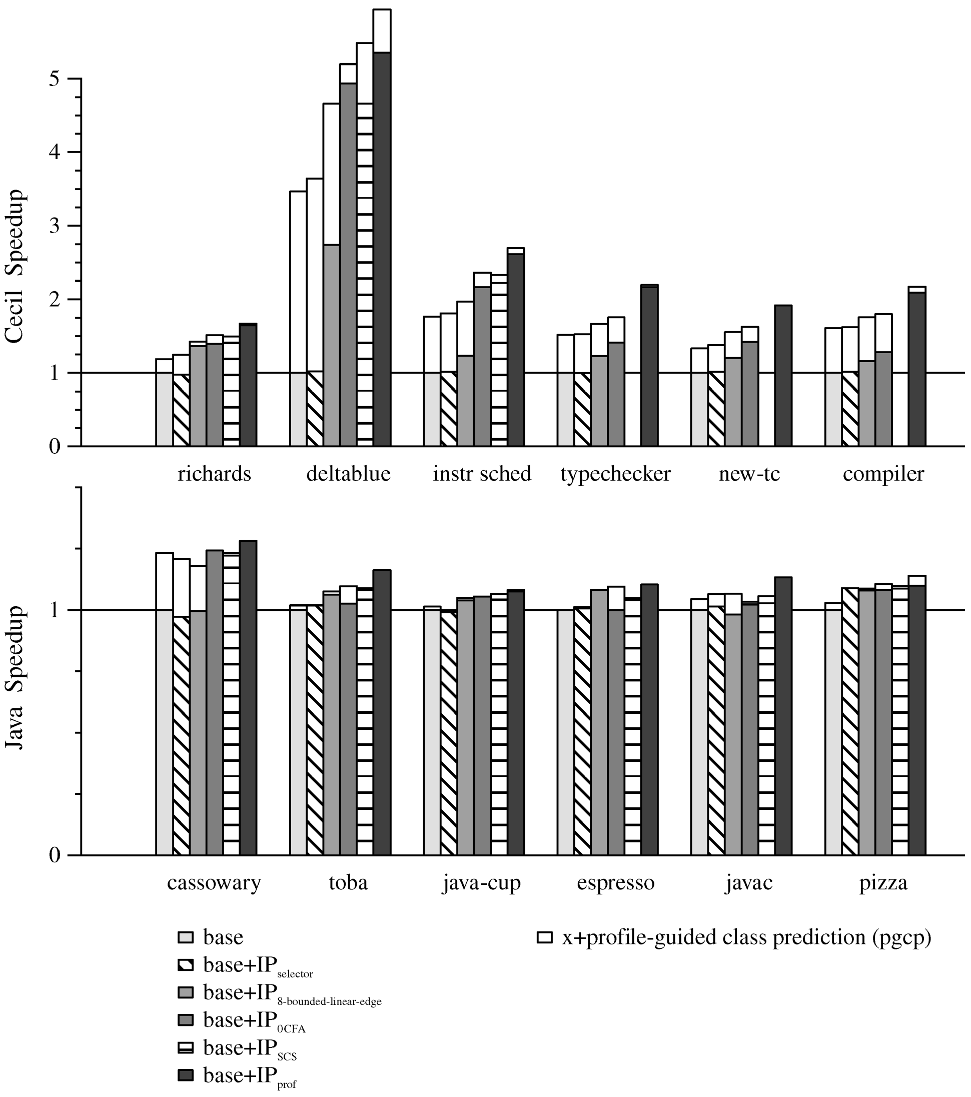


Fig. 21. Execution speed summary.

of 0-CFA for that program. Only points whose x and y values are between 0 and 200 are 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 plot is 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 three largest Cecil programs, the points corresponding to 8-Bounded Linear-Edge are clustered part way up the

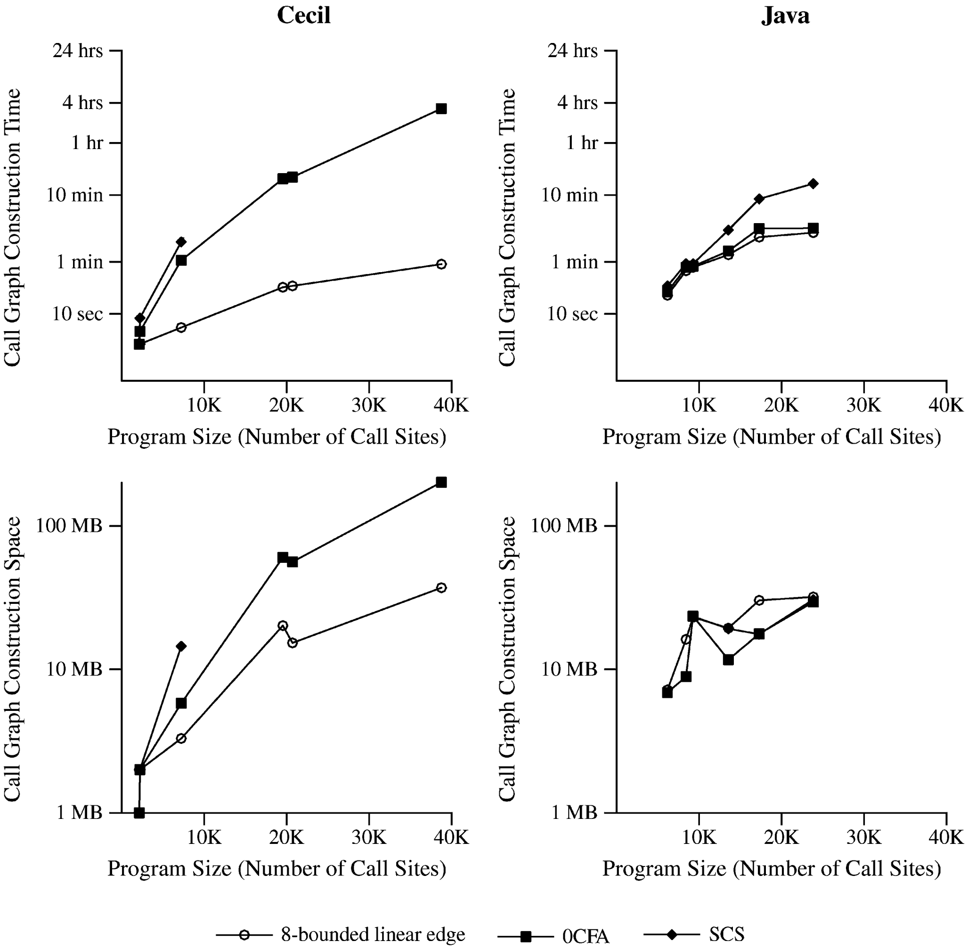


Fig. 22. Call graph construction costs summary.

y-axis very close to 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 tradeoff between 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.

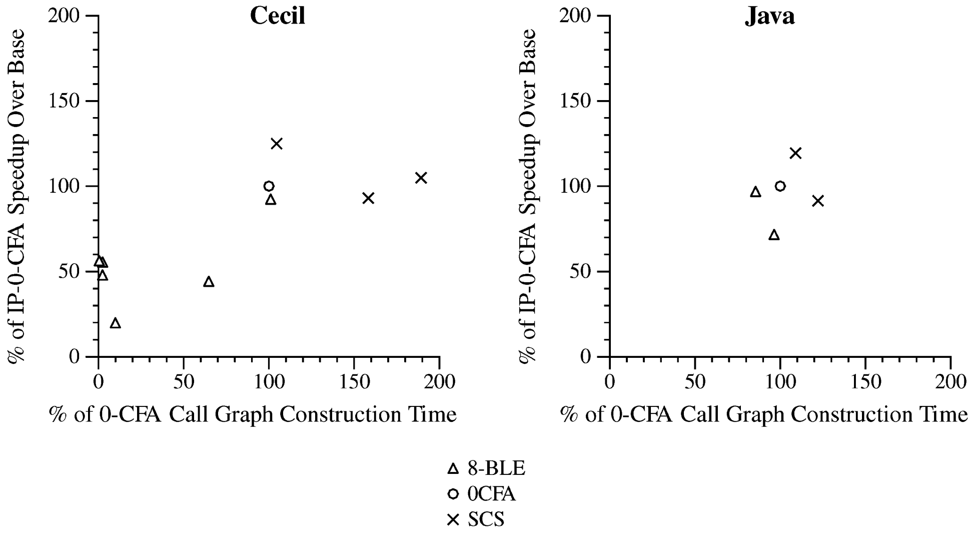


Fig. 23. Cost/benefit tradeoffs of call graph construction algorithms.

12. OTHER RELATED WORK

Closely related to our analysis framework are four parameterized control flow analyses for higher-order functional languages [Stefanescu and Zhou 1994; Jagannathan and Weeks 1995; Nielson and Nielson 1997; Palsberg and Pavlopoulou 1998]. Like our framework, all of these frameworks are parameterized to allow them to express a variety of context-sensitive analyses. Due to 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 $l > 0$, which can be expressed in ours. 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 (algorithms using equality and bounded inclusion constraints) and pessimistic algorithms. On the other hand, the analysis framework of Nielson and Nielson [1997] includes rules for analyzing explicit let-bound polymorphism, and thus can be instantiated to express polymorphic splitting [Wright and Jagannathan 1998], an analysis which is not directly expressible in our analysis framework. The analysis framework of Nielson and Nielson [1997] also includes additional parameterization over environments, which play a similar role as the class contour selection function at closure creation sites in our analysis. Unlike the other analysis frameworks, our framework is implemented in an optimizing compiler and utilized as the basis for an extensive empirical evaluation of a number of call graph construction algorithms.

Polymorphic splitting [Wright and Jagannathan 1998] 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 4.1. The analysis in Section 4.4 does not define how to analyze them. However, our analysis framework could be extended to support polymorphic splitting by incorporating the rules for let-expressions used by Wright and Jagannathan.

Agesen used *templates* as an informal explanatory device in his description of constraint-graph-based instantiations of several interprocedural class analysis algorithms [Agesen 1994]. 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.

Our analysis framework encompasses only those algorithms that monotonically converge on their final solution. A different class of algorithms, sometimes referred to as "iterative" algorithms, do not behave this way, but rather iterate phases of monotonic and nonmonotonic behavior. In previous work, we informally defined an algorithmic framework that encompassed both monotonic and nonmonotonic call graph construction algorithms [Grove et al. 1997]. However, only monotonic algorithms are currently supported in the Vortex implementation framework. In the terminology of our previous work, the analysis framework of this article is a precise definition of the Monotonic Refinement component of a more 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-execute portions of the analysis to compute a more precise final call graph. Iterative algorithms are derived from Shivers's proposal for reflow analysis [Shivers 1991]. Currently, the only iterative algorithm that has actually been implemented is Plevyak's iterative algorithm [Plevyak and Chien 1994; Plevyak 1996]. In previous work, we proposed one possible approach for designing a less aggressive iterative algorithm based on *exact unions* [Grove et al. 1997; Grove 1998].

A number of papers proposing new call graph construction algorithms have empirically assessed the effectiveness of their algorithm 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 others simply compare against a baseline system which performs no interprocedural analysis. One empirical assessment of interprocedural analysis that is somewhat different 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 and Agesen 1996]. 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.

13. CONCLUSIONS

In this article we presented a unifying framework for understanding call graph construction algorithms and an empirical comparison of a representative set of algorithms. Our parameterized call graph construction 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. Using the implementation of the general algorithm in the Vortex compiler infrastructure, we empirically evaluated a number of call graph construction algorithms on a suite of sizeable object-oriented programs. In comparison to previous experimental studies, our experiments cover a much wider range of call graph construction algorithms and include applications that are an order of magnitude larger than the largest work prior to ours. For many of these applications, interprocedural analysis enabled substantial speedups over an already highly optimized baseline. Furthermore, a significant fraction of these speedups can be obtained through the use of a scalable, near-linear-time call graph construction algorithm.

We assessed call graph construction algorithms in the context of supporting effective interprocedural analysis in an optimizing compiler for object-oriented languages. Our 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 our study is relevant both to the call graph construction algorithms and more generally to the empirical assessment of any interprocedural analysis. It is critical to evaluate algorithms on large programs, since doing so may lead to quite different conclusions than if only small programs are considered. Finally, call graphs can be utilized by a variety of other common programming environment tools. The experimental data on call graph construction costs could be used to guide the selection of the appropriate call graph construction algorithm for inclusion in such tools.

ACKNOWLEDGMENTS

This research was supported in part by an NSF grant (CCR-9503741), an NSF Young Investigator Award (CCR-9457767), and gifts from Sun Microsystems, IBM, Xerox PARC, Object Technology International, Edison Design Group, and Pure Software. The careful reading and helpful comments of Susan Eggers, David Notkin, and Jeffrey Dean greatly improved the quality of this work. The detailed comments and suggestions of the TOPLAS referees further clarified the presentation of this work. Greg DeFouw designed and implemented the initial prototype of the unification-based call graph construction algorithms of Section 10.

REFERENCES

AGESEN, O. 1994. Constraint-based type inference and parametric polymorphism. In *Proceedings of the First International Static Analysis Symposium.* LNCS 864, Springer, New York, NY, 78–100.

AGESEN, O. 1995. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Proceedings of the 9th European Conference on Object-Oriented Programming*. LNCS 952, Springer, New York, NY, 2–26.

AGESEN, O. 1996. Concrete type inference: Delivering object-oriented applications. Ph.D. thesis, SLMI Tech. Rep. 96–52, Stanford Univ., Stanford, CA.

AGESEN, O., PALSBERG, J., AND SCHWARTZBACH, M. I. 1993. Type inference of Self: Analysis of objects with dynamic and multiple inheritance. In *Proceedings of the 7th European Conference for Object-Oriented Programming*. LNCS 707, New York, NY, 247–267.

AIKEN, A. 1994. Set constraints: Results, applications, and future directions. In *Proceedings of the Second Workshop on the Principles and Practice of Constraint Programming* (Orcas Island, WA), 171–179.

ALT, M. AND MARTIN, F. 1995. Generation of efficient interprocedural analyzers with PAG. In *Proceedings of the Second International Symposium on Static Analysis*. LNCS 983, Springer, New York, NY, 33–50.

ASHLEY, J. M. 1996. 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*. ACM, New York, 184–194.

ASHLEY, J. M. 1997. The effectiveness of flow analysis for inlining. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming. ACM SIGPLAN Not. 32*, 8 (June), 99–111.

BACON, D. F. AND SWEENEY, P. F. 1996. Fast static analysis of C++ virtual function calls. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications. ACM SIGPLAN Not. 31*, 10 (Oct.), 324–341.

BAIRAGI, D., KUMAR, S., AND AGRAWAL, D. P. 1997. Precise call graph construction for OO programs in the presence of virtual functions. In *Proceedings of the 1997 International Conference on Parallel Processing*. 412–416.

CALLAHAN, D., CARLE, A., HALL, M. W., AND KENNEDY, K. 1990. Constructing the procedure call multigraph. *IEEE Trans. Softw. Eng. 16*, 4, 483–487.

CHAMBERS, C. 1993. The Cecil language: Specification and rationale. Tech. Rep. UW-CSE-93-03-05, Dept. Computer Science and Engineering. Univ. of Washington. Revised, March 1997.

CHAMBERS, C., DEAN, J., AND GROVE, D. 1996. Whole-program optimization of object-oriented languages. Tech. Rep. UW-CSE-96-06-02, Dept. of Computer Science and Engineering. Univ. of Washington. June.

CHAMBERS, C. AND UNGAR, D. 1989. 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. ACM SIGPLAN Not. 24*, 7 (July), 146–160.

CHAMBERS, C. AND UNGAR, D. 1990. 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. ACM SIGPLAN Not. 25*, 6 (June), 150–164.

COUSOT, P. AND COUSOT, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*. ACM, New York, NY, 238–252.

DEAN, J. 1996. Whole program optimization of object-oriented languages. Ph.D. thesis, Tech. Rep. UW-CSE-96-11-05, Univ. of Washington.

DEAN, J., DEFOUW, G., GROVE, D., LITVINOV, V., AND CHAMBERS, C. 1996. Vortex: An optimizing compiler for object-oriented languages. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications. ACM SIGPLAN Not. 31*, 10 (Oct.), 83–100.

Dean, J., Grove, D., and Chambers, C. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference for Object-Oriented Programming*. LNCS 952, Springer, New York, NY.

DeFouw, G., Grove, D., and Chambers, C. 1998. Fast interprocedural class analysis. In *Proceedings of the Conference Record of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, 222–236.

Deutsch, L. P. and Schiffman, A. M. 1984. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, NY, 297–302.

Diwan, A., Moss, E., and McKinley, K. 1996. Simple and effective analysis of statically-typed object-oriented programs. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications. ACM SIGPLAN Not. 31*, 10 (Oct.), 292–305.

Emami, M., Ghiya, R., and Hendren, L. J. 1994. 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. ACM SIGPLAN Not. 29*, 6 (June), 242–256.

Fernandez, M. F. 1995. Simple and effective link-time optimization of Modula-3 programs. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation. ACM SIGPLAN Not. 30*, 6 (June), 103–115.

Gosling, J., Joy, B., and Steele, G. 1996. *The Java Language Specification*. Addison-Wesley, Reading, MA.

Grove, D. 1995. The impact of interprocedural class analysis on optimization. In *Proceedings CASCON '95* (Toronto, Canada), 195–203.

Grove, D. 1998. Effective interprocedural optimization of object-oriented languages. Ph.D. thesis, Univ. of Washington.

Grove, D., Dean, J., Garrett, C., and Chambers, C. 1995. Profile-guided receiver class prediction. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications. ACM SIGPLAN Not. 30*, 10 (Oct.), 108–123.

Grove, D., DeFouw, G., Dean, J., and Chambers, C. 1997. Call graph construction in object oriented languages. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications. ACM SIGPLAN Not. 32*, 10 (Oct.), 108–124.

Hall, M. W. and Kennedy, K. 1992. Efficient call graph analysis. *ACM Lett. Program. Lang. Syst. 1*, 3 (Sept.), 227–242.

Heintze, N. and McAllester, D. 1997. Linear-time subtransitive control flow analysis. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation. ACM SIGPLAN Not. 32*, 6 (June), 261–272.

Henglein, F. 1991. Efficient type inference for higher-order binding-time analysis. In *Functional Programming and Computer Architecture*.

Hölzle, U. and Agesen, O. 1996. Dynamic vs. static optimization techniques for object-oriented languages. *Theor. Pract. Object Syst. 1*, 3.

Hölzle, U. and Ungar, D. 1994. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation. ACM SIGPLAN Not. 29*, 6 (June), 326–336.

Jagannathan, S. and Weeks, S. 1995. A unified treatment of flow analysis in higher-order languages. In *Proceedings of the Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, 393–407.

Johnson, R. E., Graver, J. O., and Zurawski, L. W. 1988. TS: An optimizing compiler for Smalltalk. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications. ACM SIGPLAN Not. 23*, 10 (Oct.).

Kam, J. B. and Ullman, J. D. 1976. Global data flow analysis and iterative algorithms. *J. ACM 23*, 1 (Jan.), 158–171.

Kildall, G. A. 1973. A unified approach to global program optimization. In *Proceedings of the Conference Record of the First ACM Symposium on Principles of Programming Languages*. ACM, New York, NY, 194–206.

Kranz, D. 1988. Orbit: An optimizing compiler for scheme. Ph.D. thesis, Res. Rep. 632, Yale Univ., Dept. of Computer Science.

Lakhotia, A. 1993. Constructing call multigraphs using dependence graphs. In *Proceedings of the Conference Record of the Twentieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, 273–284.

Landi, W. and Ryder, B. G. 1991. Pointer-induced aliasing: A problem classification. In *Proceedings of the Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, NY, 93–103.

Nielson, F. and Nielson, H. R. 1997. Infinitary control flow analysis: A collecting semantics for closure analysis. In *Proceedings of the Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, 332–345.

Oxhøj, N., Palsberg, J., and Schwartzbach, M. I. 1992. Making type inference practical. In *Proceedings ECOOP '92*, O. L. Madsen, Ed., LNCS 615, Springer, New York, NY, 329–349.

Palsberg, J. and Pavlopoulou, C. 1998. From polyvariant flow information to intersection and union types. In *Proceedings of the Conference Record of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, 197–208.

Palsberg, J. and Schwartzbach, M. I. 1991. Object-oriented type inference. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications. ACM SIGPLAN Not. 26*, 10 (Oct.), 146–161.

Pande, H. D. and Ryder, B. G. 1994. Static type determination for C++. In *Proceedings of Sixth USENIX C++ Technical Conference*.

Phillips, G. and Shepard, T. 1994. Static typing without explicit types. Unpublished report, Dept. of Electrical and Computer Engineering, Royal Military College of Canada, Kingston, Ont., Canada.

Plevyak, J. 1996. Optimization of object-oriented and concurrent programs. Ph.D. thesis, Univ. of Illinois at Urbana-Champaign.

Plevyak, J. and Chien, A. A. 1994. Precise concrete type inference for object-oriented languages. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications. ACM SIGPLAN Not. 29*, 10 (Oct.), 324–340.

Ryder, B. 1979. Constructing the call graph of a program. *IEEE Trans. Softw. Eng. 5*, 3, 216–225.

Shapiro, M. and Horwitz, S. 1997. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM, New York, NY, 1–14.

Sharir, M. and Pnueli, A. 1981. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, S. S. Muchnick and N. D. Jones, Eds. Prentice-Hall, Englewood Cliffs, NJ, Ch. 7, 189–233.

Shivers, O. 1988. Control-flow analysis in Scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation. ACM SIGPLAN Not. 23*, 7 (June), 164–174.

Shivers, O. 1991. Control-flow analysis of higher-order languages. Ph.D. thesis, Tech. Rep. CMU-CS-91-145, Carnegie Mellon Univ., Pittsburgh, PA.

Steensgaard, B. 1994. A polyvariant closure analysis with dynamic abstraction. Unpublished manuscript.

Steensgaard, B. 1996. Points-to analysis in almost linear time. In *Proceedings of the Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, 32–41.

Stefanescu, D. and Zhou, Y. 1994. An equational framework for the flow analysis of higher-order functional programs. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*. ACM, New York, NY, 190–198.

Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E., and Godin, C. 2000. Practical virtual method call resolution for Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications. ACM SIGPLAN Not. 35*, 10 (Oct.), 264–280.

Tarjan, R. E. 1975. Efficiency of a good but not linear set union algorithm. *J. ACM 22*, 2, 215–225.

Tip, F. and Palsberg, J. 2000. Scalable propagation-based call graph construction algorithms. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications. ACM SIGPLAN Not. 35*, 10 (Oct.), 281–293.

Vitek, J., Horspool, N., and Uhl, J. 1992. Compile time analysis of object-oriented programs. In *Proceedings of the Fourth International Conference on Compiler Construction*. LNCS 641, Springer, New York, NY, 237–250.

Wilson, R. P. and Lam, M. S. 1995. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation. ACM SIGPLAN Not. 30*, 6 (June), 1–12.

Wright, A. K. and Jagannathan, S. 1998. Polymorphic splitting: an effective polyvariant flow analysis. *ACM Trans. Program. Lang. Syst. 20*, 1 (Jan.), 166–207.

Received March 2000; revised June 2001; accepted July 2001