Fast Interprocedural Class Analysis

Greg DeFouw, David Grove, and Craig Chambers

Department of Computer Science and Engineering
University of Washington

{gdefouw,grove,chambers}@cs.washington.edu

Abstract

Previous algorithms for interprocedural control flow analysis of higher-order and/or object-oriented languages have been described that perform propagation or constraint satisfaction and take O(N^3) time (such as Shivers's 0-CFA and Heintze's set-based analysis), or unification and take O($N\alpha(N,N)$) time (such as Steensgaard's pointer analysis), or optimistic reachability analysis and take O(N) time (such as Bacon and Sweeney's Rapid Type Analysis). We describe a general parameterized analysis framework that integrates propagation-based and unification-based analysis primitives and optimistic reachability analysis, whose instances mimic these existing algorithms as well as several new algorithms taking O(N), O($N\alpha(N,N)$), O(N^2), and O($N^2\alpha(N,N)$) time; our O(N) and O($N\alpha(N,N)$) algorithms produce more precise results than the previous algorithms with these complexities. We implemented our algorithm framework in the Vortex optimizing compiler, and we measured the cost and benefit of these interprocedural analysis algorithms in practice on a collection of substantial Cecil and Java programs.

1 Introduction

Interprocedural class analysis computes a set of classes for each program variable, such that each run-time value bound to a variable is a direct instance of one of the classes computed for the variable. A program call graph is constructed as a side-effect of this analysis, where the classes associated with the arguments to a dynamically dispatched message send call site determine the set of callee methods that may be invoked by that call site. First-class functions and call sites of computed functions can be analyzed using interprocedural class analysis by treating each definition of a first-class function (e.g., a lambda expression) as a class with a method named `apply`, each evaluation of a first-class function definition as a class instantiation operation, and each application of a first-class function as sending the `apply` message to the function object.

A number of algorithms have been described for performing interprocedural class analysis (perhaps under different names) in object-oriented and higher-order languages. Most algorithms incrementally construct the program's dataflow graph (either implicitly or explicitly) and propagate sets of classes forward through the dataflow graph, iterating analysis in the face of loops and recursion as new call edges are discovered and new edges are added to the dataflow graph. A classic example of such an algorithm is Shivers's 0-CFA control flow analysis for Scheme [Shivers 88, Shivers 91], which in the worst case takes O(N^3) time, where N is the size of the program. Heintze's set-based analysis has a similar flavor (and complexity) to 0-CFA [Heintze 94]. Many more-expensive algorithms have been developed that include some degree of context-sensitivity or polyvariance to achieve greater precision [Oxhøj *et al.* 92, Agesen *et al.* 93, Plevyak & Chien 94, Stefanescu & Zhou 94, Agesen 95, Jagannathan & Weeks 95, Nielson & Nielson 97], but less-expensive algorithms are relatively rare. Steensgaard describes an O($N\alpha(N,N)$) pointer analysis algorithm that partitions the program's dataflow graph using unification in place of propagation, as in type inference [Steensgaard 96]; Steensgaard's algorithm was inspired by Henglein's non-standard type-inference algorithm for higher-order binding-time analysis [Henglein 91]. Bacon and Sweeney describe Rapid Type Analysis (RTA), an O(N) algorithm for optimistically removing unreachable code, which performs no propagation or unification at all [Bacon & Sweeney 96]. Heintze and McAllester describe a subtransitive version of 0-CFA that requires only O(N) time, but it applies only to statically typed programs with bounded-size types [Heintze & McAllester 97].

We have developed a general framework for interprocedural class analysis of both statically and dynamically typed programs that integrates propagation and unification. A particular dataflow analysis algorithm instantiates this general framework by specifying when and how to apply unification in place of propagation, and by specifying how many edges are used to connect call sites to callees. Instantiations of our framework include 0-CFA, Steensgaard-style analysis, and RTA, as well as interesting new algorithms with complexities of O($N^2\alpha(N,N)$), O(N^2), O($N\alpha(N,N)$) (which achieves better precision than Steensgaard-style analysis with the same worst-case cost), and O(N) (which achieves better precision than RTA with the same worst-case cost). Section 2 describes our general framework and defines and compares several algorithm instantiations.

We have implemented our algorithm framework and several instantiations in the Vortex optimizing compiler [Dean *et al.* 96]. We analyzed several large Java [Gosling *et al.* 96] and Cecil [Chambers 93] programs using these instantiations. We measured both the abstract precision and cost of the different algorithms as well as the bottom-line execution speedup and executable space savings. We found that the hypothetical improvements in precision of the new algorithms over RTA and Steensgaard-style analysis did occur in practice; resulting in improvements in bottom-line application performance. Section 3 reports our experimental findings in detail. Section 4 identifies some areas of current and future work, section 5 discusses additional related work, and section 6 concludes.

POPL 98 San Diego CA USA
Copyright 1998 ACM 0-89791-979-3/98/ 01..$3.50

```
Program        ::= {Decl} {Stmt} Expr
Decl           ::= ClassDecl | VarDecl | MethodDecl
ClassDecl      ::= class ClassID { {InstVarDecl} }
InstVarDecl    ::= instvar InstVarID
VarDecl        ::= var VarID
MethodDecl     ::= method MsgID ( {Formal} ) { {VarDecl} {Stmt} Expr }
Formal         ::= FormalID @ ClassID
Stmt           ::= LValue := Expr
Expr           ::= LValue | FormalID | NewExpr | SendExpr
NewExpr        ::= new ClassID
SendExpr       ::= send MsgID ( {Expr} )
LValue         ::= VarID | InstVarLValue
InstVarLValue  ::= Expr . InstVarID
```

Figure 1: Abstract Syntax for Example Object-Oriented Language

2 Analysis Framework

This section describes the general interprocedural analysis algorithm that allows us to explore a range of fast interprocedural class analyses. The next subsection introduces the example language we use to illustrate our algorithm. Subsection 2.2 describes our dataflow graph representation, subsection 2.3 describes the parameterized analysis algorithm itself, and subsection 2.4 analyzes its complexity. Subsection 2.5 describes the analysis algorithms instantiable from our framework. Subsection 2.6 discusses extensions to make the analysis modular. Subsection 2.7 describes how clients can extract information from the analysis, and examines the complexity of extracting certain kinds of information.

2.1 Source Language

Figure 1 shows the abstract syntax of a simple, dynamically typed, object-oriented language that we will use to help explain our framework.* It includes declarations of 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, and dynamically dispatched message sends.

A multimethod has a list of immutable formals. Each formal is specialized by a class, meaning that the method is only applicable to message sends whose actuals are instances of the corresponding specializing class or its subclasses. We assume the presence of a root class from which all other classes inherit, and specializing on this class allows a formal to apply to all arguments. Multimethods of the same name and number of arguments are related by a partial order, with one multimethod more specific than (i.e., overriding) another if its tuple of specializing classes is more specific than the other (pointwise). When a message is sent, the set of multimethods with the same name and number of arguments is collected, and, of the subset that are applicable to the actuals of the message, the unique most-specific multimethod is selected and invoked (or an error is reported if there is no such method).

Other realistic language features can be viewed as special versions of these basic features. For example, regular procedures and procedure calls can be modeled with methods none of whose formals are specialized, and literals of a particular class can be modeled with corresponding class instantiation operations (at least as far as class analysis is concerned). As described in the introduction, a first-class lexically nested function can be modeled with a class containing an `apply` method, assuming that some suitable renaming of identifiers has taken place and that local and formal variables in the lexically enclosing method can be referenced from within the `apply` method.

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

2.2 Dataflow Graph Representation

All the algorithms supported by our framework operate over a dataflow graph, declared in pseudocode in Figure 2. Figures 4 and 5 contain the algorithm for constructing the initial dataflow graph from the program being analyzed.

2.2.1 Nodes and Edges

The heart of the dataflow graph representation is a set of nodes (instances of `Node`) linked by a set of directed edges (`Edge`). Each source variable declaration, method declaration, class instantiation operation, and message send in the program has an associated node in the dataflow graph. Interprocedural class analysis computes a set of classes for each node (the `classes` member of `Node`), indicating for the corresponding variable or expression what classes of objects may be stored in the variable or returned by the expression at run time.

Two nodes are connected by a directed edge whenever classes that reach the first node can flow directly to the second node. For example, to model an assignment `target := source`, an edge is added from the node corresponding to `source` to the node corresponding to `target`. An edge may have an associated filter class set (`filter`), which restricts propagation along that edge to only classes contained in the filter set. Filters are used to restrict propagation of classes to a formal argument node of a callee method to those that are subclasses of the argument's specializing class (if given). Filters also can encode constraints ensured by static type declarations or inference, which (given the approximations that fast algorithms need to make) may make the information computed by interprocedural class analysis more precise.

2.2.2 Node Merging and Supernodes

A key feature of our framework is the ability to support merging nodes in the dataflow graph to achieve faster analysis. Our framework is parameterized by P, the maximum number of times a node may be visited during propagation; P may be any integer value between 0 and N, inclusive.† After a node has been visited P times during analysis, it is merged with each of its successor nodes.

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

† By N here we mean some value that is $O(N)$ but bigger than the number of classes in the program.

```
class SuperNode {
    rep:SuperNode;                  equivalence-class representative, initially itself
    live_nodes:set of Node;         set of active nodes in supernode, initially a single node
    dead_nodes:set of Node;         set of collapsed nodes in supernode, initially empty
    to_do:bag of ClassID;           bag of classes remaining to be processed by supernode, initially empty
    done :bag of ClassID;           bag of classes that have been processed by supernode, initially empty
}
class Node {
    super:SuperNode;                enclosing supernode
    edges:set of Edge;              set of outgoing edges
    classes:set of ClassID;         set of classes processed by this node, initially empty
    counter:int;                    number of times node can be processed before collapsing, initially P
}
class Edge {
    source, target:Node;            source and target nodes
    filter:set of ClassID;          filter of classes that can propagate across edge
}
class BarrierEdge subclass of Edge {
    barrier:Barrier;                the barrier of which this edge is a member
    blocked:bag of ClassID;         bag of classes blocked at this barrier edge, initially empty
    is_arg:bool;                    whether this is an argument edge that can release the barrier
}
class Barrier {
    edges:list of BarrierEdge;      the edges in the barrier
    num_blocked:int;                the number of barrier edges that are still blocked
    method:MethodDecl;              method that is guarded by the barrier
}
```

Figure 2: Dataflow Graph Representation

Each node records the remaining number of times it may be visited during propagation (`counter`), initialized to P. If P=0, then a node cannot be examined at all during propagation, causing nodes to be merged eagerly as connecting edges are inserted.

We introduce supernodes to represent the set of nodes that have been merged together (`SuperNode`). Supernodes partition the nodes of the graph. Initially, each node has its own unique supernode. Merging a node with its successor nodes is implemented using supernodes by unifying the supernodes corresponding to the node and its successor nodes, putting all the nodes together as members of the new unified supernode, and then "collapsing" the original node out of the dataflow graph by moving it to a separate inactive list in the unified supernode; Figure 3 illustrates merging nodes. Later, when a class is propagated to any member of the unified supernode, it is immediately forwarded to all of the active members of the supernode, skipping the inactive members, ensuring that inactive nodes never incur additional work. We use fast union-find data structures [Tarjan 75] to support quickly unifying two arbitrary supernodes and (lazily) updating all the member nodes to refer to the new unified supernode (in $O(U\alpha(U,U)+F)$ time for U unifications and F find-representative updates). To achieve unification and update in only $O(U+F)$ time, our framework allows algorithms to choose to always unify supernodes with a distinguished global supernode; our framework's *MergeWithGlobal* parameter flag selects this asymptotically faster though less precise behavior.*

Each supernode data structure refers (perhaps indirectly) to the supernode representing the unified supernode (`rep`). The representative supernode records up-to-date lists of active (`live_nodes`) and merged (`dead_nodes`) member nodes, and conversely each node refers to its containing supernode (`super`) (from which the representative supernode can be found). (The

* We include the *MergeWithGlobal* option mostly to simulate previous algorithms such as RTA.

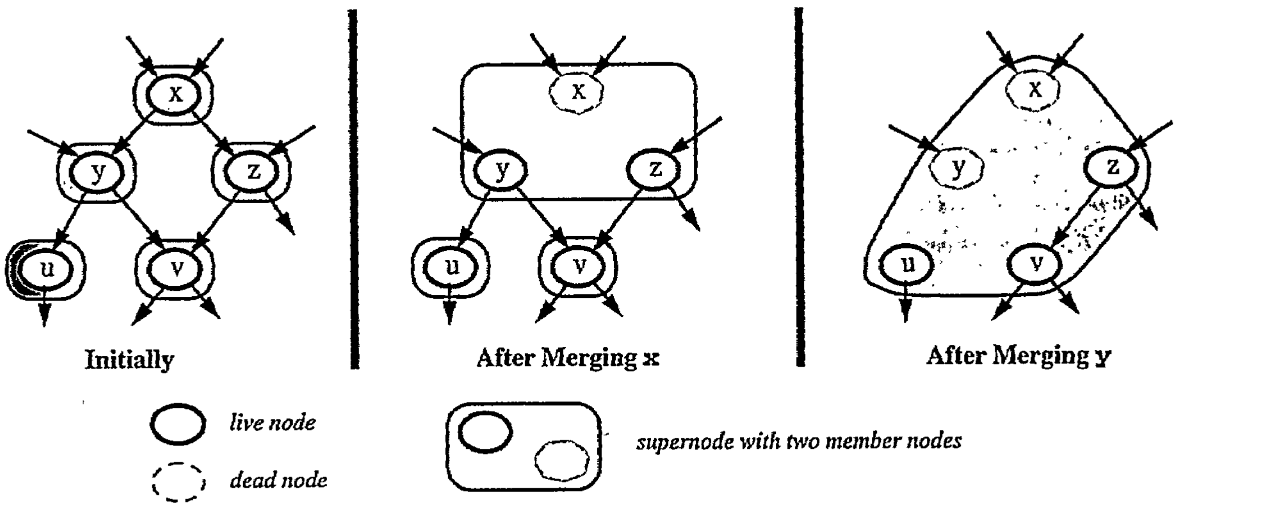

Figure 3: Example of Node Merging

```
P:int;                          parameter defining maximum number of times a node can be visited
MergeWithGlobal:bool;           parameter defining whether nodes merge with the global supernode
MergeCalls:bool;                parameter defining whether all senders of a given message are merged
nodes:set of Node;              the set of nodes in the graph
supernodes:set of SuperNode;    the set of representatives of supernodes in the graph
global:SuperNode;               a special supernode, used for MergeWithGlobal
ConstructDataflowGraph() {
    make the global node and supernode:
    global_node:Node := MakeNode();
    global := global_node.super;
    create nodes for global variables, instance variables, method formals, and method results:
    foreach ast:(VarDecl ∪ InstVarDecl ∪ MethodDecl ∪ Formal) in top-level decls do
        n:Node := MakeNode();
        ast.corresponding_node := n;
    create top-level statement and expression nodes and edges:
    CreateNodesAndEdges({}, top-level stmts, top-level expr);
}
CreateNodesAndEdges(vars:list of VarDecl, stmts:list of Stmt, expr:Expr) {
    make the nodes:
    foreach ast:(VarDecl ∪ NewExpr ∪ SendExpr) in vars ∪ stmts ∪ expr do
        ast.corresponding_node := MakeNode();
    add assignment edges:
    foreach stmt:Stmt = ⟦lv := e⟧ in stmts do
        source:Node := CorrespondingNode(e);
        target:Node := CorrespondingNode(lv);
        MakeEdge(source, target);
    construct call edges:
    foreach send:SendExpr = ⟦send msg($e_1$,...,$e_n$)⟧ in stmts ∪ expr do
        foreach i in [1..n] do
            $actual_i$:Node := CorrespondingNode($e_i$);
        node:Node := CorrespondingNode(send);
        LinkSend(msg, n, [$actual_1$,...,$actual_n$], node);
    update worklist and reachable classes from class instantiation nodes:
    foreach new:NewExpr = ⟦new c⟧ in stmts ∪ expr do
        n:Node := CorrespondingNode(new);
        AddToWorklist(n, c);
        MakeClassReachable(c);
}
CreateMethodNodesAndEdges(method:MethodDecl = ⟦method msg(...) {vars stmts expr}⟧) {
    if method has not been created yet then
        CreateNodesAndEdges(vars, stmts, expr);
}
MakeNode()→Node {
    n:Node := new Node;
    s:SuperNode := new SuperNode;
    n.super := s; n.edges := {}; n.classes := {}; n.counter := P;
    s.rep := s; s.live_nodes := {n}; s.dead_nodes := {}; s.to_do := {}; s.done := {};
    add n to nodes; add s to supernodes;
    return n;
}
MakeEdge(source, target:Node) {
    e:Edge := new Edge;
    InitEdge(e, source, target);
    InstallEdge(e);
}
InitEdge(e:Edge, source, target:Node) {
    e.source := source; e.target := target;
    e.filter := FilterFor(source) ∩ FilterFor(target);
}
InstallEdge(e:Edge) {
    add e to e.source.edges;
    if e.source.counter = 0 then
        CollapseNode(e.source);
}
MakeBarrierEdge(source,target:Node, is_arg:bool, b:Barrier)→BarrierEdge {
    e:BarrierEdge := new BarrierEdge;
    InitEdge(e, source, target);
    e.is_arg := is_arg; e.barrier := b; e.blocked := {};
    add e to b.edges;
    return e;
}
MakeBarrier(n:int, m:MethodDecl)→Barrier {
    b:Barrier := new Barrier;
    b.edges := {}; b.num_blocked := n; b.method := m;
    return b;
}
```

Figure 4: Dataflow Graph Construction Algorithm, Part 1

```
FilterFor(n:Node)→set of ClassID {
    if n created from f in Formal = ⟦v @ c⟧ then
        return set of c and its subclasses;
    if n created from VarDecl or InstVarDecl or Method and
            decl has static type T then
        return set of all classes that conform to T;
    return set of all classes;
}
CorrespondingNode(e:Expr)→Node {
    if e in VarID = ⟦varID⟧ then
        return CorrespondingVarDecl(varID).corresponding_node;
    if e in FormalID = ⟦formalID⟧ then
        return CorrespondingFormal(formalID).corresponding_node;
    if e in InstVarLValue = ⟦e' . instVarID⟧ then
        return CorrespondingInstVarDecl(instVarID).corresponding_node;
    return e.corresponding_node;
}
LinkSend(msg:MsgID, n:int, [actual$_1$:Node,...,actual$_n$:Node], result:Node) {
    if MergeCalls then
        ([msg_formal$_1$:Node,...,msg_formal$_n$:Node], msg_result:Node) :=
            MakeSharedMessageNodes(msg, n);
        foreach i in [1..n] do
            MakeEdge(actual$_i$, msg_formal$_i$);
        MakeEdge(msg_result, node);
    else
        RecordCallSite(msg, n, [actual$_1$,...,actual$_n$], result);
}
Table mapping message keys to shared message formal and result nodes, only for MergeCalls:
shared_message_nodes:(MsgID,n:int)→([Node$_1$,...,Node$_n$], Node);
MakeSharedMessageNodes(msg:MsgID, n:int)→([Node$_1$,...,Node$_n$], Node) {
    if shared_message_nodes(msg,n) not defined then
        foreach i in [1..n] do
            formal$_i$:MsgNode := MakeNode();
        result:Node := MakeNode();
        RecordCallSite(msg, n, [formal$_1$,...,formal$_n$], result);
        shared_message_nodes(msg,n) := ([formal$_1$,...,formal$_n$], result);
    return shared_message_nodes(msg,n);
}
RecordCallSite(msg:MsgID, n:int, [actual$_1$:Node,...,actual$_n$:Node], result:Node) {
    go through all the method declarations that this could map to, and create barrier links from call site to callee:
    foreach method:MethodDecl = ⟦method msg'(f$_1$@c$_1$,...,f$_{n'}$@c$_{n'}$) {...}⟧
            where msg' = msg and n' = n do
        create a tuple of barrier edges linked together in a barrier:
        barrier:Barrier := MakeBarrier(n, method);
        foreach i in [1..n] do
            formal$_i$:Node := CorrespondingNode(f$_i$);
            formal_edge$_i$:BarrierEdge := MakeBarrierEdge(actual$_i$, formal$_i$, true, barrier);
        method_result:Node := CorrespondingNode(method);
        result_edge:BarrierEdge := MakeBarrierEdge(method_result, result, false, barrier);
        link barrier edges into graph:
        foreach i in [1..n] do
            if c$_i$ in live_classes then
                add to call site now:
                InstallEdge(formal_edge$_i$);
            else
                record for later processing:
                add formal_edge$_i$ to delayed_edges(c$_i$);
        InstallEdge(result_edge);
}
live_classes:bitset of ClassID;   set of classes live in program
delayed_edges:ClassID→bag of BarrierEdge;   table mapping classes to lists of delayed edges
MakeClassReachable(c:ClassID) {
    if c ∉ live_classes then
        add c to live_classes;
        foreach edge:BarrierEdge in delayed_edges(c) do
            InstallEdge(edge);
        foreach c':ClassID in superclasses of c do
            MakeClassReachable(c');
}
```

Figure 5: Dataflow Graph Construction Algorithm, Part 2

to_do and done fields of a supernode are temporary state maintained during analysis.)

2.2.3 Optimistic Elimination of Unreachable Classes and Procedures

Our framework optimistically prunes unreachable classes and procedures, in the style of RTA [Bacon & Sweeney 96]. A method becomes reachable (and its body added to the dataflow graph) only when, for each class *C* on which one of the method's formals are specialized, a class instantiation operation for *C* or a subclass of *C* has been seen in code already known to be reachable. Several mechanisms are used in our dataflow graph representation to support optimistic pruning of unreachable code:

- A global set of reachable classes (`live_classes`) is updated as class instantiation operations are processed

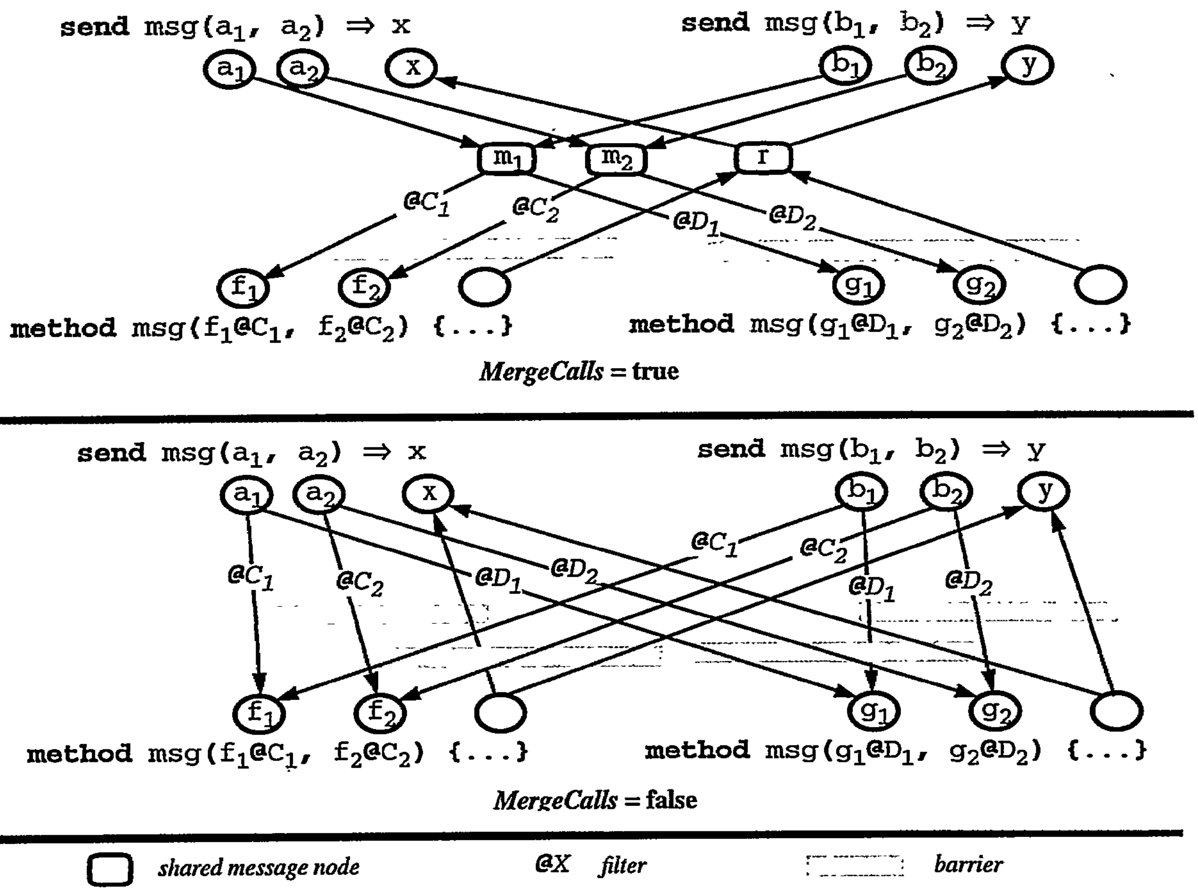

Figure 6: Example of Shared and Unshared Message Send Linkages

(MakeClassReachable). Whenever a class becomes reachable, all of its superclasses are considered reachable.

- When connecting the node for an actual parameter at a call site to the corresponding formal parameter of a callee (in RecordCallSite), only if the formal parameter's specializer class is reachable is the connection made. If not reachable, then the edge is saved on a separate list indexed by the specializer class (delayed_edges) to be entered into the dataflow graph when the specializer class becomes reachable (MakeClassReachable).
- A method specialized on reachable classes is reachable from a particular call site only if each of the actual-to-formal argument edges for that call site has a non-empty set of classes that pass through the edge's filter. To block the flow of classes through any of a call site's argument edges (and through the reverse result edge) until all the argument edges have non-empty sets of classes flowing successfully through them, we link the argument and result edges into a barrier (Barrier). A barrier records all the edges in the barrier (edges), the method that it guards (method), and a count of the number of members of the barrier that are still empty (num_blocked), initialized to the number of arguments of the method. Each time an argument edge in the barrier becomes non-empty, the barrier's blocked count is decremented. When it reaches zero, the barrier is broken and classes freely pass through the edge. A special kind of edge (BarrierEdge) is used for edges in barriers. A barrier edge knows which barrier it is a member of (barrier), and, until the barrier is broken, queues up each class that flows through the edge on a list (blocked) without forwarding it to the edge's target node. A flag (is_arg) distinguishes barrier edges that may be waited upon to become non-empty (the argument edges) from those that simply are blocked by the emptiness of other edges (the result edges).

2.2.4 Message Send Linkage

Our framework supports two approaches to connecting call sites to callee methods. If the parameter flag *MergeCalls* is false, then each actual parameter at each call site is linked to the corresponding formals of all methods with the same name and number of arguments as the call site, and the reverse for message results, leading to $O(N^2)$ edges in the dataflow graph. If *MergeCalls* is true, an intermediate tuple of nodes is created for each distinct message name and number of arguments (shared_message_nodes), one node per argument and result of the message. Actuals at call sites are linked to the corresponding intermediate message formals, which in turn are linked to the corresponding formals of the possible methods with matching name and number of arguments, and the reverse for message results, leading to only $O(N)$ edges in the graph. Figure 6 illustrates these two situations.

2.3 Parameterized Analysis Algorithm

Pseudocode for our general algorithm for interprocedural class analysis appears in Figures 7 and 8. The core of the algorithm performs propagation of classes through the dataflow graph. During the propagation phase, each supernode maintains an associated bag of classes that have reached the supernode but have not yet been processed by the supernode (to_do), as well as a bag of classes that have been processed by the supernode (done); at the end of analysis, the processed classes are used to determine the final set of classes associated with all nodes in that supernode.

```
worklist:set of SuperNode;              the set of supernodes that have non-empty to-do lists
PerformInterproceduralClassAnalysis() {
    worklist := {};
    ConstructDataflowGraph();
    ProcessWorklist();
}
ProcessWorklist() {
    while worklist non-empty do
        pop s:SuperNode off worklist;
        ProcessSuperNode(s);
}
ProcessSuperNode(s:SuperNode) {
    while FindRep(s).to_do non-empty do
        remove c:ClassID from FindRep(s).to_do;
        add c to FindRep(s).done;
        foreach n:Node in FindRep(s).live_nodes do
            ProcessNode(n, c);
        foreach n:Node in FindRep(s).live_nodes.copy do
            if n.counter = 0 then
                CollapseNode(n);
}
ProcessNode(n:Node, c:ClassID) {
    if c ∉ n.classes then
        add c to n.classes;
        foreach e:Edge in n.edges do
            ProcessEdge(e, c);
    decrement n.counter;
}
ProcessEdge(e:Edge, c:ClassID) {
    if c ∈ e.filter then
        if e is a BarrierEdge then
            ProcessBarrierEdge(e, c)
        else
            AddToWorklist(e.target, c);
}
ProcessBarrierEdge(e:BarrierEdge, c:ClassID) {
    if e.blocked is empty then
        UnblockBarrierEdge(e);
    if e.barrier.num_blocked = 0 then
        AddToWorklist(e.target, c);
    else
        add c to e.blocked;
}
UnblockBarrierEdge(e:BarrierEdge) {
    if e.barrier.num_blocked > 0 and e.is_arg then
        decrement e.barrier.num_blocked;
        if e.barrier.num_blocked = 0 then
            ReleaseBarrier(e.barrier);
}
ReleaseBarrier(b:Barrier) {
    CreateMethodNodesAndEdges(b.method);
    foreach e:BarrierEdge in b.edges do
        foreach c:ClassID in e.blocked do
            AddToWorklist(e.target, c);
}
```

Figure 7: Interprocedural Class Analysis Algorithm, Part 1

Whenever a class instantiation node is created, the instantiated class is added to the to-do list of the node's supernode.

A global worklist is maintained holding all supernodes with non-empty to-do lists (`worklist`). Our algorithm starts by constructing the nodes and edges of the top-level variable declarations, statements, and expressions in the program (`ConstructDataflowGraph`), which adds supernodes to the worklist for all the top-level class instantiation expressions. The main loop of the propagation phase (`ProcessWorklist`) removes a supernode from the worklist and processes it. The propagation phase ends when the worklist is empty (and hence all supernodes have empty to-do lists).

To process a supernode (`ProcessSuperNode`), each of the classes on its to-do list are removed one-by-one, saved on the done list, and forwarded to each of the unmerged member nodes for processing. To process a class at a member node (`ProcessNode`), if the class has not been seen at that node before, then it is propagated along to each outgoing edge of the node, and its counter of allowable future visits is decremented. To propagate a class along an edge (`ProcessEdge`), if the class passes the edge's filter, then, if the edge is not a barrier edge, the propagated class is added to the to-do list of the target node's supernode (`AddToWorklist`) which may cause the target supernode to be added to the worklist.

If the edge is a barrier edge, then there are several steps to perform (`ProcessBarrierEdge`). First, if this edge is an argument edge that is part of a blocked barrier, and this is the first class to reach this edge, then the barrier's blocked count is decremented (`UnblockBarrierEdge`). If this edge was the last edge blocking the barrier, then the barrier is broken (creating the guarded method's dataflow graph if it hasn't been created already), and all suspended classes on all edges in the barrier are released and propagated to their target supernodes (`ReleaseBarrier`). After the effect on the barrier of a class passing the edge's filter has

```
AddToWorklist(n:Node, c:ClassID) {
    if FindRep(n.super).to_do is empty then
        add FindRep(n.super) to worklist;
    add c to FindRep(n.super).to_do;
}
CollapseNode(n:Node) {
    if MergeWithGlobal then
        MergeSuperNodes(global, FindRep(n.super));
    foreach e:Edge in n.edges do
        CollapseEdge(e);
    remove n from FindRep(n.super).live_nodes;
    add n to FindRep(n.super).dead_nodes;
}
CollapseEdge(e:Edge) {
    if e is a BarrierEdge then
        CollapseBarrierEdge(e);
    MergeSuperNodes(FindRep(e.source.super), FindRep(e.target.super));
}
CollapseBarrierEdge(e:BarrierEdge) {
    if e.barrier.num_blocked > 0 and e.is_arg then
        decrement e.barrier.num_blocked;
        if e.barrier.num_blocked = 0 then
            ReleaseBarrier(e.barrier);
        else
            remove this edge from barrier
            CreateMethodNodesAndEdges(e.barrier.method);
            foreach c:ClassID in e.blocked do
                AddToWorklist(e.target, c);
            remove e from e.barrier.edges;
}
MergeSuperNodes(s1, s2:SuperNode) {
    if s1 ≠ s2 then
        rep:SuperNode := Union(s1, s2);
        rep.live_nodes := s1.live_nodes ∪ s2.live_nodes;
        rep.dead_nodes := s1.dead_nodes ∪ s2.dead_nodes;
        rep.to_do := s1.to_do ∪ s2.to_do;
        rep.done  := s1.done ∪ s2.done;
        s1.rep := rep; s2.rep := rep;
        if s1 = rep then remove s2 from supernodes
                    else remove s1 from supernodes;
}
Fast union-find data structure operations:
FindRep(s:SuperNode)→SuperNode {
    find and return the representative of the union, caching results for amortized O(α(N,N)) cost:
    if s.rep ≠ s then s.rep = FindRep(s.rep);
    return s.rep;
}
Union(s1, s2:SuperNode)→SuperNode {
    pick and return one of s1 or s2 to elect as the representative of the union; if either is global then choose it
}
```

Figure 8: Interprocedural Class Analysis Algorithm, Part 2

been computed, the class is either saved on the edge's suspended classes list (if the barrier is still blocked), or propagated through the barrier to the target supernode (if the barrier is broken).

If $P<N$, then a node's counter may reach zero, at which point it will be merged with its successor edges. After passing a class off the to-do list to a supernode's unmerged member nodes (in `ProcessSuperNode`), if a node's counter has dropped to zero, the node is merged with its successors (`CollapseNode`). To do this, the node's supernode is merged with the supernodes of each of the node's successor nodes (`CollapseEdge`), and then the node is moved from the supernode's list of unmerged members to the list of merged members, ensuring that the node will never again be examined during propagation. Merging two supernodes (`MergeSuperNodes`) selects one supernode to be the representative of the union (using the fast union-find algorithm) and combines the two supernodes' member node, to-do, and done lists. Some algorithms perform a simpler, asymptotically faster merging of supernodes, where all merging supernodes are first merged with the global supernode; the parameter flag *MergeWithGlobal* selects this behavior. If a blocked barrier edge is collapsed, that edge becomes unblocked.

2.4 Complexity Analysis

The main components of cost in our algorithm are constructing the dataflow graph (lazily), propagating classes through the dataflow graph, and merging supernodes.

2.4.1 Core Data Structures

Before examining the complexity of the main components of the algorithm, we list our assumptions about the properties of its core data structures:

- The sets of classes `SuperNode.live_nodes` and `SuperNode.dead_nodes` support constant-time initialization, set union, element addition, and element removal. To support these operations, our implementation exploits the invariant that at each step in the algorithm every instance of the `SuperNode` class is a member of at most one `live_node` or `dead_node` set. Thus, these sets can be represented by linking `SuperNodes` together in doubly linked lists.
- The bags of classes `SuperNode.to_do`, `SuperNode.done`, and `BarrierEdge.blocked` support constant-time initialization, union (ignoring duplicates), and element addition. Similarly, the bags of edges

Node.edges and Barrier.edges support constant-time initialization and element addition. Our implementation uses singly linked, circular lists to represent bags.

- The set of classes Node.classes supports constant-time initialization, membership testing, and element addition. Depending on the value of P, our implementation uses one of two representations: if P is O(1) list sets are used, while if P is $O(N)$ bit sets are used. A list set can be initialized in constant time, and it supports constant-time membership testing and element addition if the maximum size of the set is bounded by a constant. A bit set supports constant-time membership testing and element addition, but requires $O(N)$ time to initialize.
- The filter Edge.filter can be initialized in constant time and supports constant-time membership testing. The filter can be represented as a procedure to perform the subclass testing, for which there are several constant-time algorithms [AK *et al.* 89].

2.4.2 Dataflow Graph Construction

In the worst case, all classes and methods in the original program will be reachable, implying that $O(N)$ ASTs must be represented in the dataflow graph. Each kind of AST node contributes O(1) nodes to the dataflow graph. With the exception of SendExpr, each kind of AST also contributes O(1) edges. Let M (defined below) be an upper bound on the number of edges contributed by a single SendExpr AST. Then the dataflow graph contains $O(N)$ nodes and $O(N+N{\cdot}M)$ edges. Each edge in the dataflow graph can be initialized in constant time (each edge has one filter, participates in at most one barrier, and is added to one node's bag of edges). Depending on the value of P, each node takes either O(1) or $O(N)$ time to initialize. Thus the total time to construct the dataflow graph is $O(N{\cdot}M)$ if P is O(1) and $O(N^2{+}N{\cdot}M)$ if P is $O(N)$.

The value of M is either O(1) or $O(N)$ depending on the value of *MergeCalls*:*

- If *MergeCalls* is true, then an intermediate tuple of nodes (one tuple per message name) is inserted between callers and callees. Exactly one edge per actual parameter is added between a SendExpr and the corresponding node in the intermediate tuple. Similarly, the return value of the call is represented by adding one edge from the tuple's return value to the SendExpr. In addition, the intermediate tuples introduce edges connecting intermediate nodes to method formal parameters and returns; each formal parameter and method return will have exactly one such edge. These additional $O(N)$ edges are can be amortized over the $O(N)$ SendExprs in the program, thus M is O(1). This results in a total of $O(N)$ edges in the dataflow graph.
- If *MergeCalls* is false, then each SendExpr may be directly connected to $O(N)$ target methods, causing M to be $O(N)$. This results in a total of $O(N^2)$ edges in the dataflow graph.

Figure 6 illustrated these two cases.

To support lazy construction of the program dataflow graph, additional overhead is incurred to track live_classes and delayed_edges. Since a class can only become reachable once, this overhead takes $O(N{+}N{\cdot}M)$ time.

2.4.3 Propagation

If there is no SuperNode merging, the core unit of work in the propagation phase can be viewed from the perspective of a class flowing across an edge: start with a class that is new to the edge's source node (at the call to ProcessEdge in ProcessNode), and attempt to propagate it through the edge's filter. If it passes the filter, check if this is a blocked barrier edge, and if so suspend the class at the barrier edge, later to be released when the barrier is broken. Finally, add the class to the target supernode's to-do list, later remove it from the target supernode's to-do list, and then test whether it is new to the supernode's one target node (ending at the same loop of calls to ProcessEdge). Each of these steps takes constant time. By ensuring that each class is processed across an edge at most once (by maintaining Node.classes), the total amount of time for this edge propagation is $O(E{\cdot}C)$, where E is the number of edges and C is the number of classes. C is proportional to the program size (N), and E is $O(N{\cdot}M)$ as determined above (either $O(N)$ or $O(N^2)$, depending on the value of *MergeCalls*), leading to a total cost for edge propagation of $O(N^2{\cdot}M)$ time. The time to visit each supernode on the worklist and start the edge propagation process is $O(N)$, leading to an overall time for propagation of $O(N^2{\cdot}M)$.

2.4.4 Unification

If $P < N$, then some nodes may be collapsed during propagation or graph construction. This affects the complexity of analysis in three ways: the number of times a node (and consequently its successor edges) may be visited is reduced from N to P, additional work to collapse nodes is incurred, and the calls to FindRep may take more time due to collapsing.

- Instead of using the Node.classes set to bound the number of times a node is visited by the number of classes ($O(N)$), node collapsing bounds the number of times a node is visited by P. Under this model, the constant-time unit of work sequence is slightly shifted, since now multiple nodes may be in a supernode: start with considering a member node of a supernode for a particular class at the call to ProcessNode inside ProcessSuperNode, then follow the class flowing through the node and an edge through to being added and then later removed from a supernode's to-do list, ending at the same loop of calls to ProcessNode. Each of these constant-time units of work may only be done P times per edge. Using P in place of one N in the time for edge propagation gives a more general complexity assessment of $O(P{\cdot}N{\cdot}M)$.
- Each call to MergeSuperNodes and CollapseEdge takes constant time, and each call to CollapseNode takes constant time ignoring the per-edge work subsumed by CollapseEdge, leading to an overall cost for node collapsing of $O(N{\cdot}M)$ time.
- If *MergeWithGlobal* is false, then the calls to FindRep can now take more than constant time, but overall, given a fast union-find data structure implementation of supernodes, the additional cost for all of the FindRep calls is $O(N\alpha(N,N))$. If *MergeWithGlobal* is true, however, all of the supernodes merge directly with the global supernode, preserving the constant-time behavior of FindRep.

2.4.5 Summary

Overall, the complexity of the entire graph construction and propagation phase is thus $O(P{\cdot}N{\cdot}M + N{\cdot}M)$, plus $O(N{\cdot}M\alpha(N,N))$ if $P<N$ and *MergeWithGlobal* is false. By setting P to some constant, new algorithms with worst-case time complexities of $O(N)$, $O(N^2)$, $O(N\alpha(N,N))$, and $O(N^2\alpha(N,N))$ result, depending on the choices for *MergeWithGlobal* and *MergeCalls*.

* We assume that the maximum number of actual parameters at a call site and the maximum number of formal parameters in a method declaration is a constant independent of program size.

Table 1: Framework Instantiations

Algorithms	*P*	*MergeWithGlobal*	*MergeCalls*	*Complexity*
Classic OO 0-CFA	N	n/a	false	$O(N^3)$
Linear-Edge OO 0-CFA	**N**	**n/a**	**true**	$O(N^2)$
Bounded OO 0-CFA	**O(1)**	**false**	**false**	$O(N^2\alpha(N,N))$
Bounded Linear-Edge OO 0-CFA	**O(1)**	**false**	**true**	$O(N\alpha(N,N))$
Simply Bounded OO 0-CFA	**O(1)**	**true**	**false**	$O(N^2)$
Simply Bounded Linear-Edge OO 0-CFA	**O(1)**	**true**	**true**	$O(N)$
Equivalence Class Analysis	0	false	true	$O(N\alpha(N,N))$
RTA	0	true	true	$O(N)$

2.5 Instantiations of the Framework

Table 1 identifies several algorithms that are instantiations of our framework; boldface rows are new algorithms.

Classic OO 0-CFA is the standard cubic-time, flow-sensitive but context-insensitive interprocedural class analysis. Equivalence Class Analysis is Steensgaard-style near-linear-time division of the program's dataflow graph into disjoint subgraphs, extended to work in the object-oriented context. RTA is Bacon and Sweeney's Rapid Type Analysis algorithm.

The five other algorithms represent new interesting points in the analysis design space. The three Linear-Edge algorithms bound the number of call edges, dropping a factor of $O(N)$ from the complexity of the other (quadratic-edge) algorithms. The two Bounded algorithms use supernodes and merging to ensure only a constant number of visits per node, dropping another factor of $O(N)$ from the complexity (but adding back in the near-constant $O(\alpha(N,N))$ to pay for the overhead of merging). The two Simply Bounded algorithms avoid this extra $O(\alpha(N,N))$ overhead by merging all supernodes with the distinguished global supernode. The Bounded Linear-Edge algorithm and the Steensgaard-style Equivalence Class Analysis have the same near-linear worst-case time complexities, but the Bounded Linear-Edge algorithm always provides solutions that are at least as precise and often more precise than Equivalence Class Analysis. Similarly, the Simply Bounded Linear-Edge algorithm incurs the same linear-time complexity but delivers precision at least as good and often better than RTA.

2.6 Analyzing Program Components

As described and implemented, our analysis framework assumes it has access to the entire program. Our framework could be extended to support more modular analyses by allowing components of programs to be modeled by summary dataflow graphs. Components whose source code is unavailable can then be analyzed as long as a summary dataflow graph is available. (The summary dataflow graph need not be precise, merely a sound approximation of the "true" dataflow graph.) Furthermore, components can be partially pre-analyzed, starting from known sources of class information within the component, with the resulting partially propagated and/or collapsed dataflow graph being used in the analysis of containing programs. This would lead to a kind of hierarchical, component-wise analysis of programs that may help the analyses scale to larger programs, along the lines of Flanagan and Felleisen's componential set-based analysis [Flanagan & Felleisen 97].

2.7 Clients of the Analysis

Our analysis provides information to clients in two forms. The program call graph can be constructed in time proportional to the number of edges by recording when barriers along call edges are broken; each such broken barrier corresponds to an edge in the program call graph. Additionally, the done list of the supernode of each variable's node records the bag of classes that may be stored in that variable. A number of interesting optimizations can exploit this information in only constant time per access, including:

- checking whether only one method can be called from a given call site and if so replacing that dynamically dispatched message with a direct procedure or inlined code,
- skipping compilation of any methods not called from any call site in the call graph (treeshaking).

Some other uses of the information may require more work, however. For example, to support constant-time testing of whether a particular class is a member of a particular variable's set of possible classes, to optimize run-time class tests for instance, the done bag for the variable's supernode needs to be converted into a set, which requires quadratic time in the worst case for algorithms with *MergeWithGlobal* false and $P<N$. (Algorithms where *MergeWithGlobal* is true can simply use the set of live classes as the classes set for the distinguished global node, and uncollapsed nodes maintain the set of classes reaching them directly.) Consequently, the bounded linear-edge algorithm, with asymptotic complexity $O(N\alpha(N,N))$, may not be appropriate for clients which require this per-variable set-of-classes information.

Other authors of sub-quadratic algorithms have also encountered difficulties providing useful information to clients. For example, Steensgaard presents a near-linear-time algorithm for performing pointer analysis, but to completely query the resulting data structure to compute all points-to relationships among variables would require quadratic time [Steensgaard 96]. But if only a subset of the possible points-to relationships are of interest, then less time may be incurred in a particular algorithm. Similarly, Heintze and McAllester describe subtransitive control flow analysis which constructs an encoded representation of the 0-CFA dataflow graph in linear time (for a restricted language model with function types bounded in size by a constant), but performing the transitive closure to compute the full explicit dataflow graph requires quadratic time [Heintze & McAllester 97]. They offer other queries of their encoded representation, such as computing the call sites which have only one callee, which require only linear time.

Table 2: Benchmark Applications

	Program	Lines[a]	Description
Cecil	richards	400	Operating systems simulation
Cecil	deltablue	650	Incremental constraint solver
Cecil	instr sched	2,400	Global instruction scheduler
Cecil	typechecker	20,000	Cecil typechecker
Cecil	compiler	50,000	Old version of Vortex compiler
Java	toba	3,900	Java bytecode to C translator
Java	espresso	13,800	Java source to bytecode translator[b]
Java	javac	25,550	Java source to bytecode translator[b]

a. Excluding standard libraries. All Cecil programs are compiled with an 11,000-line standard library. All Java programs include a 16,000-line standard library.
b. The two Java translators have no common code and were developed by different people.

3 Experimental Assessment

In addition to asymptotic complexity results, we wish to understand how well the different algorithms perform in practice. Accordingly, we implemented our framework in the Vortex optimizing compiler [Dean *et al.* 96] and applied all eight algorithms to the collection of large Cecil and Java programs described in Table 2. We assessed the algorithms according to the following three criteria:

- What are the relative precisions of the sets of classes and the induced call graph produced by the various algorithms?
- What are the relative costs of the various algorithms, measured in terms of analysis time and space costs?
- How do the differences in precision translate into differences in the bottom-line effectiveness of client optimizations, in terms of program execution speed and executable size?

The results of our experiments are shown in Figures 9 and 10.* Each graph plots two pairs of two lines, one pair for bounded and simply bounded linear-edge OO 0-CFA and one pair for bounded and simply bounded (quadratic-edge) OO 0-CFA, with P varying from 0 to N along the x-axis. When P=N, the pairs of lines converge into linear-edge OO 0-CFA and classic OO 0-CFA, respectively. In the degenerate case when P=0, bounded linear-edge OO 0-CFA is equivalent to equivalence class analysis and simply bounded linear-edge OO 0-CFA is equivalent to RTA. Subsection 3.1 discusses the measured time and space costs of analysis, Subsection 3.2 discusses the relative abstract precision of the different algorithms, and Subsection 3.3 addresses the impact of the results of interprocedural class analysis on run-time speed and executable size. All experiments were performed on a Sun Ultra-1 model 170.

3.1 Time and Space Costs

The first column of graphs shows the analysis times in seconds. Overall, asymptotic time complexity is a fairly good predictor of actual analysis time. As program size increases, the time required to perform instances of the two linear-time algorithmic families also increases linearly. The gap between the linear-time and the quadratic-time algorithms widens as program size increases. The larger constant values for P incur small increases in analysis time over P=0. For the four smallest programs, analysis time actually decreases as P grows from 50 to N; in these programs, the additional precision of the P=N configuration significantly reduces the number of reachable methods (and thus the number of nodes and edges in the dataflow graph) which compensates for the additional propagation across non-unified edges.

For the larger programs, the space cost of explicitly representing the entire dataflow graph, especially when *MergeCalls* is false, becomes prohibitive. The missing data points for the typechecker and compiler programs are due to excessive memory consumption. Future work includes implementing a more space-efficient representation of the dataflow graph, and investigating mixing partially implicit representations of the dataflow graph with node unification.

3.2 Abstract Precision

A number of metrics can be used to measure the abstract precision of interprocedural class analysis. The second and third columns of graphs present data for two of these metrics that are closely related to the optimizations performed by Vortex using the results of interprocedural class analysis.

- *Percentage of Singleton Class Sets*: Each node in the dataflow graph has an associated set of classes. What fraction of these nodes contain only a single class? This metric provides an abstract measure of the precision of interprocedural class analysis and may be indicative of how useful the information will be when it is consulted during intraprocedural class analysis.
- *Percentage of Singleton Callees*: A call graph can be built as interprocedural analysis proceeds. What fraction of all message sends in the program can be proved to only invoke one target method? This metric is closely related to the effectiveness of static binding (and subsequent inlining) of message sends during intraprocedural compilation and optimization.

Since it only maintains a single global set of classes, RTA does not have any singleton class sets. The Steensgaard-style Equivalence Class Analysis has the potential to do better than RTA, but only succeeds in doing so on a subset of the benchmark programs; in the larger Cecil programs it was unable to identify enough disjoint regions of the dataflow graph to impact the results. However, modest increases in the value of P (up to about P=5) yield large increases in the fraction of singleton class sets. We observed diminishing returns for larger constant values of P, but setting P=N results in a large increase in the percentage of singleton class sets. Increasing the number of edges from $O(N)$ to $O(N^2)$ has a negligible impact on the percentage of singleton class sets. Similar trends also hold for the percentage of singleton callees metric, with the slight complication that due to treeshaking the total number of call sites actually decreases as P increases, and thus in a few cases the percentage of singleton callees actually slightly decreases as algorithmic precision increases.

3.3 Bottom-Line Impact of Abstract Precision

To assess both the bottom-line impact of interprocedural analysis as well as how well the abstract precision metrics described in the previous subsection predict algorithmic effectiveness, we compared, for each benchmark and algorithm pair, the performance of a base configuration that did not use interprocedural optimizations against a configuration performing interprocedural optimizations building on the class sets and call graphs produced by the algorithm. The base configuration

* A longer version of this paper is available that contains the complete set of numerical data in addition to the graphs [DeFouw *et al.* 97].

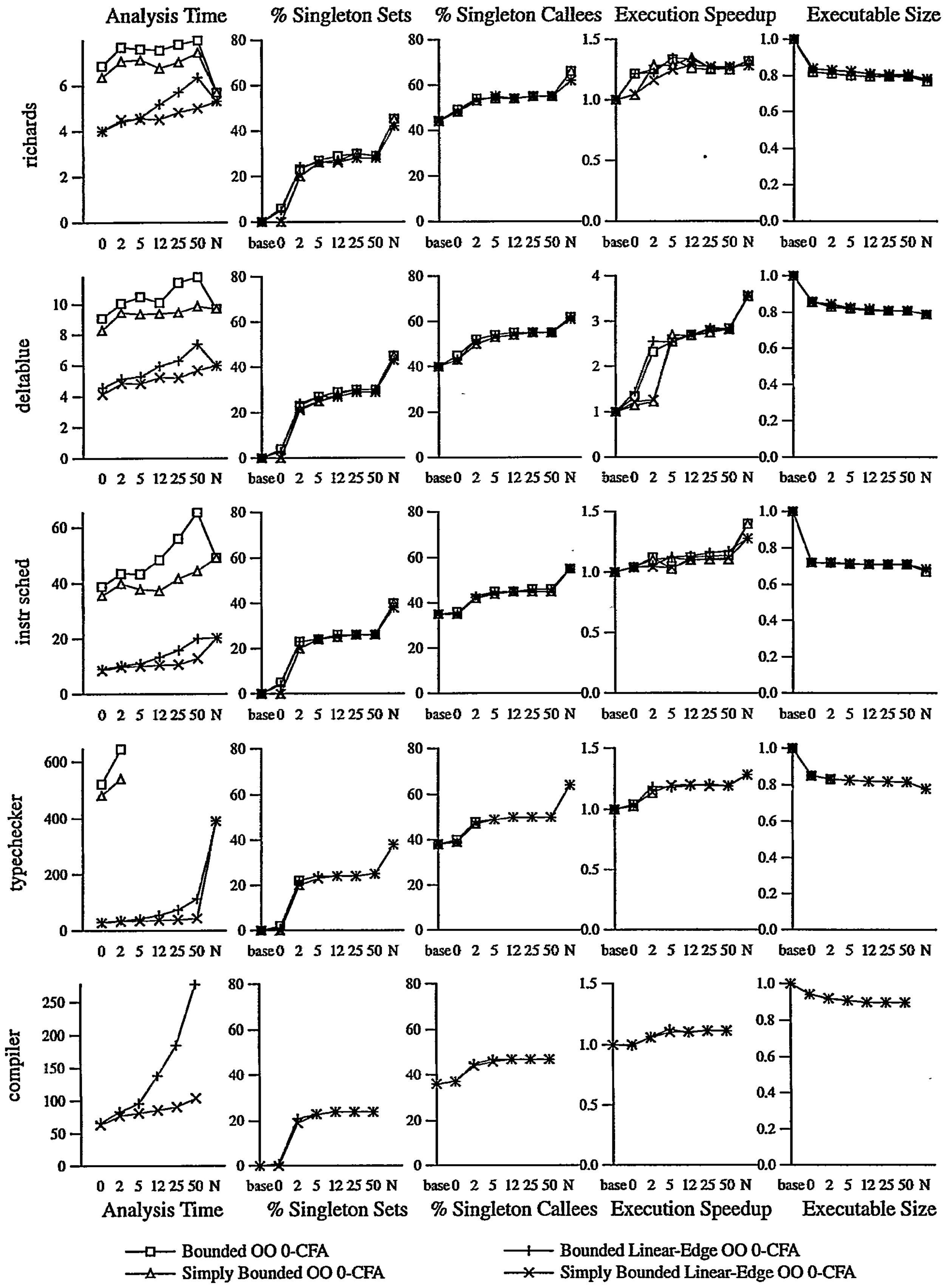

Figure 9: Experimental Results (Cecil)

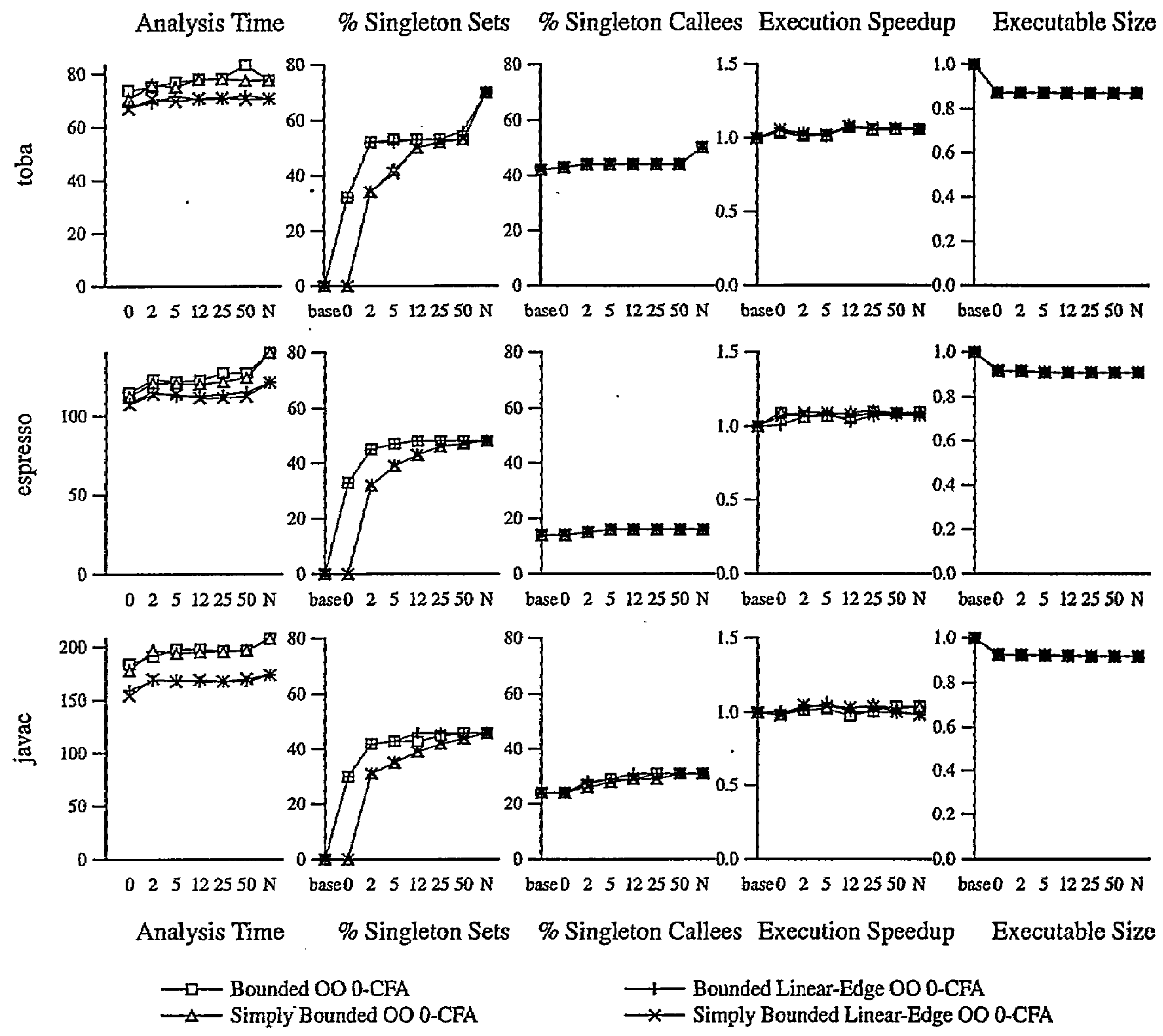

Figure 10: Experimental Results (Java)

represents an aggressive combination of intraprocedural and limited interprocedural optimizations which include: intraprocedural class analysis [Johnson 88, Chambers & Ungar 90], hard-wired class prediction for common messages (Cecil programs only) [Deutsch & Schiffman 84, Chambers & Ungar 89], splitting [Chambers & Ungar 89], whole-program class hierarchy analysis [Dean *et al.* 95], cross-module inlining, static class prediction [Dean *et al.* 96, Dean 96] and closure optimizations (Cecil only). We applied these optimizations through our Vortex compiler to produce C code, which we then compiled with `gcc -O2` to produce executable code.

The interprocedural configuration augments the base configuration with interprocedural analyses that enabled the intraprocedural optimizations in base to work better:

- Class analysis: Intraprocedural class analysis exploits the class sets and the sets of possible callee methods computed by interprocedural analysis, enabling better optimization of dynamically dispatched messages.
- Treeshaking: As a side-effect of constructing the call graph, the compiler identifies those procedures which are unreachable during any program execution. The compiler does not compile any unreachable procedures, often resulting in substantial reductions both in code size and compile time.

The final two columns of graphs present application speedups and executable sizes relative to the base configuration. Interprocedural class analysis enabled speedups ranging from marginal improvements to slightly over a factor of three speedup on one benchmark. With the exception of the two smallest benchmarks, the P=0 configurations (RTA and Equivalence Class Analysis) did not help run-time speed. Increasing the value of P beyond 0, however, improved run-time speed, and as foreshadowed by the abstract precision results, a fairly small P value (e.g., 5) was sufficient to obtain most of the benefit available for constant values of P. The additional precision obtained in the P=N configurations translated into additional performance improvements over the P=50 configurations.

The least precise algorithm (RTA) was sufficient to enable most of the reduction in executable size. Increasing values of P enable little additional improvement over RTA for most benchmarks.

The number of edges, either O(N) or O(N^2) depending on the value of *MergeCalls*, did not have a significant impact on either application speedup or executable size. Thus, the two linear-edge algorithms are clearly preferable to their quadratic-edge counterparts, since they obtain virtually identical bottom-line results while consuming less analysis time and space. For the two smallest benchmarks, the additional potential precision of the bounded algorithms *vs.* the simply bounded algorithms had an impact on bottom-line application performance, but there was not a measurable difference for the majority of the benchmarks.

4 Future Work

We are currently investigating a number of extensions to our framework for fast interprocedural class analysis. First, we are studying how to adapt the idea of lazily merging nodes, present for propagation, to apply to merging call site message nodes. Initially, each call site could get its own separate message node, but use merging to ensure that each method is reached by at most one call site. This would ensure a linear bound on the number of edges in the graph while still enabling a fair amount of separation between independent callers. This facility may be particularly helpful for invocations of closures, where the shared `apply` formal and result message nodes introduced eagerly when *MergeCalls* is true are smearing the argument and result class sets of all closures with a particular number of arguments together, while lazy merging of these message nodes could often keep closures used in simple ways isolated from one another.

Allowing a quadratic number of edges in the graph offers a kind of context-sensitive or polyvariant analysis of the virtual generic function that dispatches messages with a given name and number of arguments to the appropriate member methods. More generally, we wish to explore adding other more explicit forms of context-sensitivity to our fast analyses. In other work we have examined the impact of different context-sensitivity strategies on cost and precision of algorithms building upon the cubic-time classic OO 0-CFA algorithm [Grove *et al.* 97], but we have not considered adapting those notions to the faster algorithms presented here, nor have we considered ways of bounding the worst-case cost of context-sensitivity.

The parameters to our framework allow placing bounds on different aspects of the algorithm, to achieve better worst-case time and space costs. However, each of these bounds was ensured uniformly across the program on a local basis. An alternative approach could more adaptively redistribute the total budget of allowable work units so that parts of the graph that do not come close to the original uniform bound can redistribute their unused work units to be used in parts of the graph that are more challenging. Similarly, some kinds of approximations are more costly in final precision than others; for example, merging two nodes within a method body probably has much less negative impact on the quality of the final solution than does collapsing a barrier node which may allow whole trees of methods to become reachable that shouldn't be, incurring much more work to process the bodies of the otherwise unreachable methods.

5 Additional Related Work

Our framework integrates traditional propagation-based analyses such as 0-CFA and type-inference-style, unification-based analyses such as Steensgaard's pointer analysis, as well as coping with object-oriented method dispatch and supporting optimistic pruning of unreachable classes and methods. Ashley presents an algorithm framework parameterized by a context-sensitivity operator and an operator for removing undesired precision of abstract values during analysis [Ashley 96, Ashley 97]. He instantiates his framework to produce an algorithm that performs only a bounded amount of propagation before falling back to a distinguished Unknown abstract value, which resembles our simply bounded OO 0-CFA O(N^2) algorithm. Our framework additionally supports local unification (*MergeWithGlobal* = false), linear-edge variants, and object-oriented language features. Unlike our framework as presented here, Ashley's framework supports context-sensitive analysis, and he examined combining his bounded algorithm with 1-CFA-style context-sensitivity.

Relatively few interprocedural control flow or class analyses have been implemented and measured on substantial programs. In order of increasing asymptotic complexity of the examined algorithms, Bacon and Sweeney examined C++ programs up to 30,000 lines in size, Steensgaard examined C programs up to 25,000 lines, Ashley examined Scheme programs up to 30,000 lines in size, Agesen examined Self programs up to 7,000 lines (although all but one were 1,000 lines or smaller), and Plevyak and Chien examined Concurrent Aggregates programs up to 2,000 lines. Our benchmarks span a range from several hundred to 60,000 lines in size (including library code), enabling us to assess the scalability of the different algorithm instances beyond that examined by previous work.

6 Conclusion

We have developed a parameterized algorithm for interprocedural class analysis that describes a continuum of different algorithms ranging in cost from O(N) to O(N^3). Our framework integrates both propagation-style analysis and unification-style analysis, allowing specific algorithms to mix the two methods to achieve desired time costs and precision benefits. Since interprocedural class analysis is very similar in spirit to control flow analysis, closure analysis, and set-based analysis, and includes mechanisms found in (non-standard) type inference systems, versions of our new algorithms should be applicable in a wide range of interprocedural analysis domains for languages with data-dependent control flow (e.g., first-class functions and/or dynamic dispatching).

We have implemented this framework and measured its effectiveness on a number substantial Cecil and Java programs. The new bounded and simply bounded linear-edge OO 0-CFA algorithms substantially improve the values of such abstract metrics as the percentage of singleton class sets and singleton callees, in comparison to previous linear- and near-linear-time algorithms for interprocedural class analysis. This improvement in abstract precision often translates into improvements in bottom-line application speed and compactness.

Acknowledgments

We thank Bjarne Steensgaard for several discussions on the topic of near-linear-time interprocedural analysis. Michael Ernst, Todd Millstein, and the anonymous POPL reviewers provided helpful comments on the presentation in this paper. This research is supported in part by an NSF grant (number CCR-9503741), an NSF Young Investigator Award (number CCR-9457767), a grant from the Office of Naval Research (contract number N00014-94-1-1136), an Intel Foundation Graduate Fellowship, and gifts from Sun Microsystems, IBM, Xerox PARC, Object Technology International, Edison Design Group, and Pure Software.

References

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

[Agesen *et al.* 93] Ole Agesen, Jens Palsberg, and Michael I. Schwartzback. Type Inference of Self: Analysis of Objects with Dynamic and Multiple Inheritance. In *Proceedings ECOOP '93*, July 1993.

[AK *et al.* 89] Hassan A"it-Kaci, Robert Boyer, Patrick Lincoln, and Roger Nasr. Efficient Implementation of Lattice Operations. *ACM Transactions on Programming Languages and Systems*, 11(1):115–146, January 1989.

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

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

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

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

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

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

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

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

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

[DeFouw *et al.* 97] Greg DeFouw, David Grove, and Craig Chambers. Fast Interprocedural Class Analysis. Technical Report TR-97-07-02, Department of Computer Science and Engineering. University of Washington, July 1997.

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

[Flanagan & Felleisen 97] Cormac Flanagan and Matthias Felleisen. Componential Set-Based Analysis. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 235–248.

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

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

[Heintze & McAllester 97] Nevin Heintze and David McAllester. Linear-Time Subtransitive Control Flow Analysis. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation* [PLD97], pages 261–272.

[Heintze 94] Nevin Heintze. Set-Based Analysis of ML Programs. In *Proceedings of the ACM Conference on LISP and Functional Programming '94*, pages 306–317, Orlando, FL, June 1994.

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

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

[Johnson 88] Ralph Johnson. TS: AN Optimizing Compiler for Smalltalk. In *Proceedings OOPSLA '88*, pages 18–26, November 1988. Published as ACM SIGPLAN Notices, volume 23, number 11.

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

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

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

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

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

[Steensgaard 96] Bjarne Steensgaard. Points-to Analysis in Almost Linear Time. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* [POP96], pages 32–41.

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

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