The Dissertation Committee for Benjamin Charles Hardekopf
certifies that this is the approved version of the following dissertation:

Pointer Analysis: Building a Foundation for Effective Program Analysis

Committee:

Calvin Lin, Supervisor

Kathryn McKinley

Keshav Pingali

William Cook

Michael Hind

Pointer Analysis: Building a Foundation for Effective Program Analysis

by

Benjamin Charles Hardekopf, B.S.; M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2009

Dedicated to my mother, Celia Mullane Hardekopf.

Acknowledgments

I owe a great deal to a great many people who have helped shaped my life and my career. My greatest thanks goes to my parents, who provided me with a strong foundation growing up and instilled in me an appreciation for knowledge and learning that have stood me in good stead for my entire life. I wouldn't be where I am or who I am without them.

I would also like to thank my advisor, Calvin Lin, for giving me the support and encouragement I've desperately needed while working towards my degree. It took a long haul, with many detours along the way, but he never waivered. My other committee members, Kathryn McKinley, William Cook, Keshav Pingali, and Mike Hind, have also contributed a great deal to my development as a scientist by giving me the benefit of their time and advice. All of them, by their advice and examples, have greatly influenced my approach to computer science research and have shown me what to strive for in my own career.

Finally, I would like to thank my fellow students, both within and without my research group. Their companionship and support during these many years have helped me make it through the long slog to the very end.

Pointer Analysis: Building a Foundation for Effective Program Analysis

Publication No. _____

Benjamin Charles Hardekopf, Ph.D.
The University of Texas at Austin, 2009

Supervisor: Calvin Lin

Pointer analysis is a fundamental enabling technology for program analysis. By improving the scalability of precise pointer analysis we can make a positive impact across a wide range of program analyses used for many different purposes, including program verification and model checking, optimization and parallelization, program understanding, hardware synthesis, and more.

In this thesis we present a suite of new algorithms aimed at improving pointer analysis scalability. These new algorithms make inclusion-based analysis (the most precise flow- and context-insensitive pointer analysis) over 4× faster while using 7× less memory than the previous state-of-the-art; they also enable flow-sensitive pointer analysis to handle programs with millions of lines of code, two orders of magnitude greater than the previous state-of-the-art.

We present a formal framework for describing the space of pointer analysis approximations. The space of possible approximations is complex and multi-dimensional, and until now has not been well-defined in a formal manner. We

believe that the framework is useful as a method to meaningfully compare the precision of the multitude of existing pointer analyses, as well as aiding in the systematic exploration of the entire space of approximations.

Table of Contents

Chapter 1

Introduction

One of the most important problems facing computer science today is the sheer size and complexity of modern software. Software is becoming one of the foundations of society, and yet at the same time software is becoming more difficult to understand, more difficult to guarantee correct, and more difficult to optimize. Program analysis is a key tool to manage this complexity; it has been used for such diverse purposes as model checking [4, 42], security analysis [13, 30, 74], error-checking [34], hardware synthesis [86], software refactoring [41], and parallelization [16, 72], among many others.

While program analysis can do a great deal to help deal with software complexity, there is a fly in the ointment: software contains *indirection*, and this indirection makes program analysis both more difficult and less effective. The two main forms of indirection are *indirect data-flow* (e.g., pointer dereferences in C and object accesses in Java) and *indirect control-flow* (e.g., function pointers in C, virtual method dispatch in object-oriented languages such as Java, and closures in higher-order languages such as Lisp). While some languages don't contain explicit pointers or pointer dereferences, at a low level (such as a compiler's intermediate representation), all of these kinds of indirection are implemented using pointers. The goal of pointer analysis is to resolve this indirection by computing *points-to sets*: for each program entity (variable, object reference, etc), its points-to set is the set of memory locations that can be indirectly referenced via that entity.

Because indirection is so ubiquitous and central to programming languages, pointer analysis is a fundamental enabling technology for program analysis: in order to analyze a program's behavior and properties, the indirection present in the program must be resolved as precisely as possible (i.e., the points-to sets should be as small as possible). The more precisely the indirection is resolved, the more effective program analysis can be. By improving the precision and scalability of pointer analysis, we can directly contribute to the effectiveness of a wide variety of program analyses such as the ones listed above.

1.1 Pointer Analysis Background

Pointer analysis, like most static analyses, is an undecidable problem [49]. However, even after making common approximations which make many static analyses tractable (e.g., restricting dynamic memory and ignoring branch conditions), pointer analysis remains NP-hard [12]. Therefore, practical pointer analysis requires further approximations.

The space of possible approximations for pointer analysis is complex and multi-dimensional; some of the dimensions of precision that can be approximated include: flow-sensitivity, context-sensitivity, field-sensitivity, the heap model, representation of pointer information, branch conditions, and array indexing, among others. Pointer analysis is a very mature field, having been studied for decades with many papers published on the subject (for examples, see Hind's survey on pointer analysis research [43]). Researchers have studied many different combinations of approximations using many techniques, such as dataflow analysis [44, 50], set constraints [29, 37], type systems [25, 78], and CFL reachability [77, 84].

1.1.1 Dimensions of Precision

As pointed out above, there are many different dimensions of precision that can be modeled when approximating pointer analysis. We will focus here on two of the most important dimensions, flow- and context-sensitivity.

Flow-Sensitivity. Flow-sensitivity determines whether the analysis models the fact that a variable's value can change over time. A flow-sensitive analysis respects a program's control-flow and computes a separate solution for each program point, as opposed to a flow-insensitive analysis which ignores control-flow and computes for each variable a single solution that conservatively holds over the entire program.

Context-Sensitivity. Context-sensitivity determines whether the analysis models the fact that each separate invocation of a procedure is independent from all other invocations. A context-sensitive analysis analyzes a procedure independently for each calling context, as opposed to a context-insensitive analysis which merges all of the calling contexts together and analyzes them together.

Current State-of-the-Art. Flow- and context-sensitivity are independent of each other; an analysis can be either flow-sensitive or flow-insensitive and at the same time either context-sensitive or context-insensitive. Flow- and context-insensitive analyses are the most scalable type of pointer analysis. Inclusion-based analysis (the most precise analysis in this class) can analyze on the order of a million lines of code [41]. Adding either flow- or context-sensitivity severely degrades scalability. A context-sensitive analysis can only analyze on the order of a few hundred thousand lines of code [64], while a flow-sensitive analysis can only analyze on the order of a few tens of thousands of lines of code [44].

1.2 Thesis Contributions

This thesis makes the following contributions:

- A set of new algorithms for inclusion-based pointer analysis (the most precise flow- and context-insensitive pointer analysis) that make the analysis over 4× faster while using 7× less memory than the previous state-of-the-art.
- Two new algorithms for flow-sensitive, context-insensitive pointer analysis that make the analysis almost 200× faster while using almost 50× less memory, increasing the scalability of the analysis by two orders of magnitude (from a tens of thousands of lines of code to millions of lines of code).
- A formal framework, based on operational semantics, for describing the space of pointer analysis approximations, which makes this space well-defined and amenable to systematic exploration.

Several of these new algorithms have already had a practical impact outside the research community—they are used by a number of groups, including the GCC compiler infrastructure, the LLVM compiler infrastructure, and Semantic Designs (a company that builds software engineering tools to analyze and transform programs with tens of millions of lines of code).

1.3 Outline of Thesis

Chapter 2 describes inclusion-based analysis, the most precise of the flow- and context-insensitive analyses. The chapter gives background on the analysis and describes some related work, then details and evaluates our new algorithms for improving the scalability of inclusion-based analysis.

Chapter 3 describes flow-sensitive, context-insensitive analysis. The chapter gives background on the analysis, including why flow-sensitivity is important and what makes it so expensive, and it describes some related work on making flow-sensitive analysis scalable. Then the chapter details and evaluates our new algorithms for scalable flow-sensitive analysis.

Chapter 4 pulls back and looks at the bigger picture of pointer analysis, addressing the ill-defined and largely unexplored space of pointer analysis approximations. The chapter describes a formal framework based on operational semantics that precisely describes the precision of the majority of existing pointer analysis algorithms, something not previously possible.

Finally, Chapter 5 concludes the thesis by recapping our contributions and speculating on future work.

Chapter 2

Inclusion-based Analysis

Inclusion-based analysis is the most precise of the flow- and context-insensitive pointer analyses. This chapter describes a set of new algorithms for inclusion-based pointer analysis that significantly increase its scalability, both in terms of analysis time and memory consumption. Section 2.1 provides background on the basic concept of inclusion-based analysis; Section 2.2 describes related work for scalable inclusion-based algorithms; then Sections 2.3 and 2.4 describe our new algorithms and evaluate their performance with respect to the current state-of-the-art.

While several algorithms for inclusion-based analysis have been proposed prior to this work, the competing algorithms have never been compared head-to-head to determine their relative performance. Besides the new algorithms we describe, our empirical comparison of all the related work is another contribution of this thesis. When utilized together, our new algorithms are on average over $4\times$ faster and use over $7\times$ less memory than the best of the previous state of the art algorithms. The work described in this chapter has been previously published by Hardekopf and Lin [37] and Hardekopf and Lin [38].

2.1 Background

Inclusion-based analysis relies on two fundamental approximations to make pointer analysis tractable. The first approximation eliminates control-flow from a

program, leaving only an unordered set of assignment statements. These assignment statements include parameter assignments that soundly approximate the effects of function calls (which were removed as part of the control-flow). For example, a function call $x = foo(y)$ to a function foo with parameter p and returning value r would be replaced by the set of assignments $p = y$ and $x = r$. The result of removing all control-flow is an approximate program which can execute any assignment after any other assigment and which can execute assignments an arbitrary number of times.

The second approximation replaces each assignment with an inclusion constraint. Whereas an assignment $x = y$ means that x takes on the value of y, the new, corresponding constraint $x \supseteq y$ means that x's value includes y's value. This approximation together with the first approximation guarantees that the final analysis solution at any point in the program is identical to the solution at any other point in the program, and therefore we can safely compute a single solution for the entire program rather than a separate solution for each program point.

Since the values we're interested in for pointer analysis are points-to sets, the resulting set of constraints form a system of equations that constrain the possible points-to sets of each program variable. The goal of inclusion-based analysis is to compute the smallest points-to set for each variable such that all of the constraints are satisfied. For simplicity we will assume that all of the generated constraints are of the types shown in Table 2.1. For a variable v, $pts(v)$ represents v's points-to set and $loc(v)$ represents the memory location denoted by v.

Computing the points-to sets is done with the help of a data structure called the *constraint graph*. A constraint graph G has one node for each program variable. For each direct constraint $a \supseteq b$, G has a directed edge $b \rightarrow a$. Each node also has a points-to set associated with it, initialized using the init constraints: for each init

Constraint Type	Program Code	Constraint	Meaning
Init	$a = \&b$	$a \supseteq \{b\}$	$loc(b) \in pts(a)$
Direct	$a = b$	$a \supseteq b$	$pts(a) \supseteq pts(b)$
Indirect$_1$	$a = *b$	$a \supseteq *b$	$\forall v \in pts(b) : pts(a) \supseteq pts(v)$
Indirect$_2$	$*a = b$	$*a \supseteq b$	$\forall v \in pts(a) : pts(v) \supseteq pts(b)$

Table 2.1: Constraint Types

constraint $a \supseteq \{b\}$, node a's points-to set contains $loc(b)$. The indirect constraints are not explicitly represented in the graph; instead they are maintained in a separate list.

The edges of the constraint graph represent the constraints between variables. If an edge goes from variable x to variable y, then y's points-to set must include x's points-to set. The analysis can satisfy the constraints represented in the graph by propagating the variables' points-to sets along the edges of the graph: for each edge $x \rightarrow y$, $pts(y) \hookleftarrow pts(x)$ (where $\hookleftarrow$ represents set update). However, not all the constraints are represented in the graph: indirect constraints can't be represented because (as shown in Table 2.1) we need to know the variables' points-to sets in order to know what edges to add; since the whole point of the analysis is to compute the variables' points-to sets, this presents a quandary.

The analysis solves this quandary by dynamically adding edges to the graph during the analysis itself. As the analysis updates variables' points-to sets, it adds new edges to the graph to represent the indirect constraints that can be resolved using the new points-to information. If variable b's points-to set is updated, then for each constraint $a \supseteq *b$ and each $loc(v) \in pts(b)$, we add a new edge $v \rightarrow a$. Similarly, for each constraint $*b \supseteq a$ we add a new edge $a \rightarrow v$.

Figure 1 shows a basic worklist algorithm that maintains the explicit transi-

tive closure of G by continually propagating points-to sets along G's edges, adding new edges when appropriate. The worklist is initialized with all nodes in G that have a non-empty points-to set. The $\hookleftarrow$ operator represents set update. For each node n taken off the worklist, we proceed in two steps:

1. For each $loc(v) \in pts(n)$: for each constraint $a \supseteq *n$ add an edge $v \rightarrow a$, and for each constraint $*n \supseteq b$ add an edge $b \rightarrow v$. Any node that has had a new outgoing edge added is inserted into the worklist.

2. For each outgoing edge $n \rightarrow v$, propagate $pts(n)$ to node v, i.e., $pts(v) := pts(v) \cup pts(n)$. Any node whose points-to set has been modified is inserted into the worklist.

The algorithm is presented as it is for clarity of exposition; various optimizations are possible to improve its performance.

2.2 Related Work

Inclusion-based pointer analysis was first described by Andersen in his Ph.D. thesis [1], in which he formulates the problem in terms of type theory. The algorithm presented in the thesis doesn't use a constraint graph; instead it solves the inclusion constraints in a fairly naive manner by repeatedly iterating through a constraint vector. There have been several significant updates since that time.

Faehndrich et al. [29] were the first to represent the constraints using a graph and formulate the problem as computing the dynamic transitive closure of that graph. This work introduces the notion of *cycle detection*, an important optimization for inclusion-based analysis that will be discussed further in Section 2.3. The authors propose a method for partial online cycle detection and demonstrate

Algorithm 1 Basic inclusion-based analysis.

Require: $G = \langle N, E \rangle$, $Worklist = N$
  while $Worklist \neq \emptyset$ **do**
    $n \leftarrow$ SELECT$(Worklist)$
    for all $v \in pts(n)$ **do**
      for all constraints $a \supseteq *n$ **do**
        if $v \rightarrow a \notin E$ **then**
          $E \hookleftarrow \{v \rightarrow a\}$
          $Worklist \hookleftarrow \{v\}$
      for all constraints $*n \supseteq b$ **do**
        if $b \rightarrow v \notin E$ **then**
          $E \hookleftarrow \{b \rightarrow v\}$
          $Worklist \hookleftarrow \{b\}$
    for all $n \rightarrow z \in E$ **do**
      $pts(z) \hookleftarrow pts(n)$
      if $pts(z)$ changed **then**
        $Worklist \hookleftarrow \{z\}$

that cycle detection is critical for scalability of inclusion-based analysis. In their method, a depth-first search of the graph is performed upon every edge insertion, but the search is artificially restricted for the sake of performance, making cycle detection incomplete.

Heintze and Tardieu introduce a new algorithm for computing the dynamic transitive closure [41]. As new edges are added to the constraint graph from the indirect constraints, the new points-to information is not automatically propagated across the edges. Instead, the constraint graph retains its pre-transitive form. During the analysis, indirect constraints are resolved via reachability queries on the graph. Cycle detection is performed as a side-effect of these queries. The main drawback to this technique is unavoidable redundant work—it is impossible to know whether a reachability query will encounter a newly-added inclusion edge (inserted earlier due to some other indirect constraint) until after it completes, which means that

potentially redundant queries must still be carried out on the off-chance that a new edge will be encountered. Heintze and Tardieu report excellent results, analyzing a C program with 1.3M LOC in less than a second, but these results are for a field-based implementation. A field-based analysis treats each field of a struct as its own variable—assignments to $x.f$, $y.f$, and $(*z).f$ are all treated as assignments to a variable f, which tends to decrease both the size of the input to the pointer analysis and the number of dereferenced variables (an important indicator of performance). Field-based analysis is unsound for C programs, and while such an analysis is appropriate for the work described by Heintze and Tardieu (the client is a dependency analysis that is itself field-based), it is inappropriate for many others. For the results in this dissertation, we use a field-insensitive version of their algorithm, which is dramatically slower than the field-based version[1].

Pearce et al. have proposed two different approaches to inclusion-based analysis, both of which differ from Heintze and Tardieu in that they maintain the explicit transitive closure of the constraint graph (i.e., they propagate points-to information as in the basic algorithm given by Figure 1). Pearce et al. first proposed an analysis [67] that uses a more efficient and complete algorithm for online cycle detection than Faehndrich et al. [29]. In order to avoid cycle detection at every edge insertion, the algorithm dynamically maintains a topological ordering of the constraint graph. Only a newly-inserted edge that violates the current ordering could possibly create a cycle, so only in this case are cycle detection and topological reordering performed. This algorithm proves to still have too much overhead (mainly due to continually updating the topological order), so Pearce et al. later proposed a new and more efficient algorithm [66]. Rather than detect cycles at the point when a

[1]To ensure that the performance difference is in fact due to field-insensitivity, we also benchmarked a field-based version of our HT implementation. We observed comparable performance to that reported by Heintze and Tardieu [41].

new edge is inserted, the entire constraint graph is periodically swept to detect and collapse any cycles that have formed since the last sweep. This second algorithm is the one we compare against in our evaluation.

Berndl et al. [7] describe an inclusion-based pointer analysis for Java that uses BDDs [10] to represent both the constraint graph and the points-to solution. BDDs have been extensively used in model checking as a way to represent and manipulate large graphs in a very compact and efficient way. Berndl et al. were one of the first to use BDDs for pointer analysis. The analysis they describe is specific to the Java language; it also doesn't handle indirect function calls because it depends on a prior analysis to construct the complete call-graph. We implement a version of the algorithm for comparison in this thesis that is targeted for C programs and that does handle indirect function calls.

Rountev et al. [70] introduce Offline Variable Substitution (OVS), a linear-time offline analysis (i.e., carried out before the inclusion-based analysis), whose aim is to find and collapse pointer-equivalent variables (i.e., variables with identical points-to sets). OVS is complementary to cycle detection and is orthogonal to the actual inclusion-based algorithm used; it can be combined with any of the algorithms we've discussed. Our offline algorithms, described in Section 2.4, subsume and improve upon OVS.

Because inclusion-based analysis has in the past been considered to be non-scalable, other algorithms, including Steensgaard's near-linear time analysis [78] and Das' One-Level Flow analysis [25], have been proposed to improve performance by making further approximations and sacrificing additional precision. While Steensgaard's analysis has much greater imprecision than inclusion-based analysis, Das reports that for C programs the One-Level Flow analysis has precision very close to that of inclusion-based analysis. This precision is based on the assumption

that multi-level pointers are less frequent and less important than single-level pointers, which Das' experiments indicate is usually (though not always) true for C; this assumption may not hold true for other languages, such as Java and C++. In addition, for the sake of performance Das conservatively unifies non-equivalent variables, much like Steensgaard's analysis; this unification makes it difficult to trace dependency chains among variables. Dependency chains are very useful for understanding the results of program analyses such as program verification and program understanding, and also for use in tools such as Broadway [34]. Inclusion-based pointer analysis is a better choice than either Steensgaard's analysis or One-Level Flow, *if* it can be made to run in reasonable time even on large programs with millions of lines of code; this is the challenge that we address in this thesis.

In the other direction of increasing precision, there have been several attempts to scale a context-sensitive version of inclusion-based pointer analysis. One of the most scalable of these attempts is the algorithm by Whaley et al. [81], which uses BDDs to scale a context-sensitive inclusion-based pointer analysis for Java to roughly 600K LOC (measuring bytecode rather than source lines). However, Whaley et al.'s algorithm is only context-sensitive for top-level variables, meaning that all variables in the heap are treated context-insensitively. Its efficiency depends heavily on certain characteristics of the Java language—attempts to use the same technique for analyzing programs in C have shown greatly reduced performance [2].

Nystrom et al. [64] present a context-sensitive algorithm based on the insight that inlining all function calls makes a context-insensitive analysis equivalent to a context-sensitive analysis of the original program. Of course, inlining all function calls can increase the program size exponentially, but intelligent heuristics can help prevent exponential growth. An important building block of this approach is

context-insensitive inclusion-based analysis—it is used while inlining the functions and also for analyzing the resulting program. Nystrom et al. manage to scale the context-sensitive analysis to a C program with 200K LOC. The new techniques described in this dissertation should scale their algorithm even further.

2.3 Online Cycle Detection

Faehndrich et al. [29] observed that, given a cycle in the constraint graph, every variable in the cycle must necessarily have identical points-to sets. This observation follows because as soon as the points-to set for one variable changes, that change gets propagated to every other variable in the cycle. Because these variables are identical, they can all be collapsed together into a single node in the constraint graph without losing precision.

The important question is how to detect these cycles. Most cycles in the constraint graph aren't present in the initial graph; they only appear during the analysis as it adds new edges. This observation motivates *online* cycle detection, i.e., cycle detection conducted periodically during the course of the inclusion-based analysis. Online cycle detection has an inherent tension between aggressiveness and overhead. In one extreme, the analysis could check to see if a cycle was created each time a new edge is added to the graph; this strategy has a tremendous amount of overhead that negates any benefit of the cycle detection optimization. In the other extreme, the analysis could wait until the end of the analysis to check for cycles; however, this strategy loses any opportunity for optimizing the analysis itself. We need a strategy that finds a sweet spot between these two extremes.

The particular method used for detecting cycles will in large part determine the efficiency of the inclusion-based analysis. We now present two new approaches

for online cycle detection that balance this tension in different ways.

2.3.1 Lazy Cycle Detection

The central insight behind cycle detection is that cycles in the constraint graph can be collapsed because nodes in the same cycle are guaranteed to have identical points-to sets. We turn this fact around to create a heuristic for cycle detection: nodes with identical points-to sets might be part of a cycle. The insight is to balance aggression versus overhead by only looking for a cycle when there is evidence that a cycle might exist. Before propagating points-to information across an edge of the constraint graph, we check to see if the source and destination already have equal points-to sets; if so then we use a depth-first search to check for a possible cycle.

This technique is lazy because rather than trying to detect cycles when they are created, i.e., when the final edge is inserted that completes the cycle, it waits until the effect of the cycle—identical points-to sets—becomes evident. The advantage of this technique is that we only attempt to detect cycles when we are likely to find them. A potential disadvantage is that cycles may be detected well after they are formed, since we must wait for the points-to information to propagate all the way around the cycle before we can detect it.

The efficiency of this technique depends upon the assumption that two nodes usually have identical points-to sets only because they are in the same cycle; otherwise it would waste time trying to detect non-existent cycles. One additional refinement is necessary to bolster this assumption: we never trigger cycle detection on the same edge twice. We thus avoid making repeated cycle detection attempts involving nodes with identical points-to sets that are not in a cycle. This additional restriction implies that Lazy Cycle Detection is incomplete—it is not guaranteed to

find all cycles in the constraint graph.

The Lazy Cycle Detection algorithm is shown in Figure 2. Before we propagate a points-to set from one node to another, we check to see if two conditions are met: (1) the points-to sets are identical; and (2) we haven't triggered a search on this edge previously. If these conditions are met, then we trigger cycle detection rooted at the destination node. If there exists a cycle, we collapse together all the nodes involved; we also remember this edge so that if no cycle exists we won't repeat the attempt later.

Algorithm 2 Lazy cycle detection.

Require: $G = \langle N, E \rangle$, $Worklist = N$, $R = \emptyset$
  while $Worklist \neq \emptyset$ **do**
    $n \leftarrow$SELECT$(Worklist)$
    for all $v \in pts(n)$ **do**
      for all constraints $a \supseteq *n$ **do**
        if $v \rightarrow a \notin E$ **then**
          $E \hookleftarrow \{v \rightarrow a\}$
          $Worklist \hookleftarrow \{v\}$
      for all constraints $*n \supseteq b$ **do**
        if $b \rightarrow v \notin E$ **then**
          $E \hookleftarrow \{b \rightarrow v\}$
          $Worklist \hookleftarrow \{b\}$
    for all $n \rightarrow z \in E$ **do**
      if $pts(z) == pts(n) \wedge n \rightarrow z \notin R$ **then**
        DETECT-AND-COLLAPSE-CYCLES(z)
        $R \hookleftarrow \{n \rightarrow z\}$
      $pts(z) \hookleftarrow pts(n)$
      if $pts(z)$ changed **then**
        $Worklist \hookleftarrow \{z\}$

2.3.2 Hybrid Cycle Detection

Cycle detection can be done offline, in a static analysis prior to the actual pointer analysis, such as with Offline Variable Substitution described by Rountev et al. [70]. However, as mentioned earlier many cycles don't exist in the initial constraint graph and only appear as new edges are added during the pointer analysis itself, thus the need for online cycle detection techniques such as Lazy Cycle Detection. The drawback to online cycle detection is that it requires traversing the constraint graph multiple times searching for cycles; these repeated traversals can become extremely expensive. Hybrid Cycle Detection (HCD) is so-called because it combines both offline and online analyses to detect cycles, thereby getting the best of both worlds—detecting cycles created online during the pointer analysis, without requiring any traversal of the constraint graph.

2.3.2.1 Offline Component

The HCD offline analysis is a linear-time static analysis done prior to the actual pointer analysis. We build an offline version of the constraint graph, with one node for each program variable plus an additional *ref* node for each variable dereferenced in the constraints (e.g., $*n$). There is a directed edge for each direct and indirect constraint: $a \supseteq b$ yields edge $b \rightarrow a$, $a \supseteq *b$ yields edge $*b \rightarrow a$, and $*a \supseteq b$ yields edge $b \rightarrow *a$. Init constraints are ignored. Figure 2.1 illustrates this process.

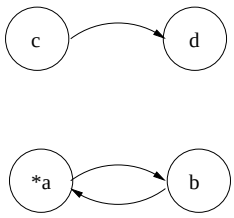
Once the graph is built we detect strongly-connected components (SCCs) using Tarjan's linear-time algorithm [79]. Any SCCs containing only non-ref nodes can be collapsed immediately. SCCs containing ref nodes are more problematic: a ref node in the offline constraint graph is a stand-in for a variable's unknown points-to set, e.g., the ref node $*n$ stands for whatever n's points-to set will be when the

$a = \&c;$
$d = c;$
$b = *a;$
$*a = b;$

(a) Program

$a \supseteq \{c\}$
$d \supseteq c$
$b \supseteq *a$
$*a \supseteq b$

(b) Constraints

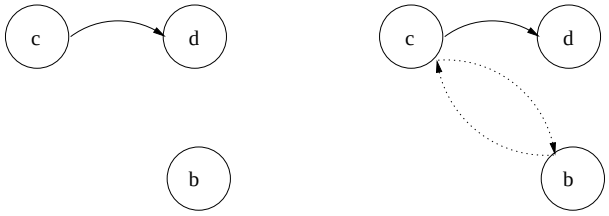


(c) Offline Constraint Graph

Figure 2.1: HCD Offline Analysis Example: (a) Program code; (b) constraints generated from the program code; (c) the offline constraint graph corresponding to the constraints. Note that $*a$ and b are in a cycle together; from this we can infer that in the online constraint graph, b will be in a cycle with all the variables in a's points-to set.

$$a \rightarrow \{c\}$$

(a) Points-to Info



(b) Before edges added **(c) After edges added**

Figure 2.2: HCD Online Analysis Example: (a) The initial points-to information from the constraints in Figure 2.1; (b) the online constraint graph before any edges are added; (c) the online constraint graph after the edges are added due to the indirect constraints in Figure 2.1. Note that *c* and *b* are now in a cycle together.

pointer analysis is complete. An SCC containing a ref node such as $*n$ actually means that n's points-to set is part of the SCC; but since we don't yet know what that points-to set will be, we can't collapse that SCC. The offline analysis knows which variables' points-to sets will be part of an SCC, while the online analysis (i.e., the pointer analysis) knows the variables' actual points-to sets. The purpose of Hybrid Cycle Detection is to bridge this gap. Figure 2.2 shows how the online analysis is affected when an SCC contains a ref node in the offline constraint graph.

We finish the offline analysis by looking for SCCs in the offline constraint graph that consist of more than one node and that also contain at least one ref node. Because there are no constraints of the form $*p \supseteq *q$, no ref node can have a reflexive edge and any non-trivial SCC containing a ref node must also contain a non-ref node. For each SCC of interest we select one non-ref node b, and for each ref node $*a$ in the same SCC, we store the tuple (a, b) in a list L. This tuple signifies

to the online analysis that a's points-to set belongs in an SCC with b, and therefore everything in a's points-to set can safely be collapsed with b.

2.3.2.2 Online Component

The online analysis is shown in Figure 3. The algorithm is similar to the basic algorithm shown in Figure 1, except when processing node n we first check L for a tuple of the form (n,a). If one is found then we preemptively collapse together node a and all members of n's points-to set, knowing that they belong to the same cycle. For simplicity's sake the pseudo-code ignores some obvious optimizations.

Algorithm 3 Hybrid cycle detection.

Require: $G = \langle N, E \rangle$, $Worklist = N$
  while $Worklist \neq \emptyset$ **do**
    $n \leftarrow$ SELECT$(Worklist)$
    for all $(n,a) \in L$ **do**
      for all $v \in pts(n)$ **do**
        COLLAPSE(v,a)
        $W \hookleftarrow \{a\}$
    for all $v \in pts(n)$ **do**
      for all constraints $a \supseteq *n$ **do**
        if $v \rightarrow a \notin E$ **then**
          $E \hookleftarrow \{v \rightarrow a\}$
          $Worklist \hookleftarrow \{v\}$
      for all constraints $*n \supseteq b$ **do**
        if $b \rightarrow v \notin E$ **then**
          $E \hookleftarrow \{b \rightarrow v\}$
          $Worklist \hookleftarrow \{b\}$
    for all $n \rightarrow z \in E$ **do**
      $pts(z) \hookleftarrow pts(n)$
      if $pts(z)$ changed **then**
        $Worklist \hookleftarrow \{z\}$

Hybrid Cycle Detection is not guaranteed to find all cycles in the online

Name	LOC	Constraints	Init	Direct	Indirect
Emacs-21.4a	169K	21,460	4,088	11,095	6,277
Ghostscript-8.15K	169,312	67,310	12,154	25,880	29,276
Gimp-2.2.8	554K	96,483	17,083	43,878	35,522
Insight-6.5	603K	85,375	13,198	35,382	36,795
Wine-0.9.21	1,338K	171,237	39,166	62,499	69,572
Linux-2.4.26	2,172K	203,733	25,678	77,936	100,119

Table 2.2: Benchmarks: For each benchmark we show the number of lines of code (computed as the number of non-blank, non-comment lines in the source files), the number of constraints generated using CIL after being optimized with OVS, and a break-down of the types of constraints.

constraint graph, only those that can be inferred from the offline version of the graph. Those cycles that it does find, however, are discovered at the earliest possible opportunity and without requiring any traversal of the constraint graph. In addition, while HCD can be used on its own as shown in Figure 3, it can also be easily combined with other cycle detection mechanisms, such as LCD, to enhance their performance.

2.3.3 Evaluation

To compare the various inclusion-based pointer analyses, we implement field-insensitive versions of five main algorithms: Heintze and Tardieu (HT), Berndl et al. (BLQ), Pearce et al. (PKH), Lazy Cycle Detection (LCD), and Hybrid Cycle Detection (HCD). We also implement four additional algorithms by integrating HCD with the other four main algorithms: HT+HCD, PKH+HCD, BLQ+HCD, and LCD+HCD. The algorithms are written in C++ and handle all aspects of the C language except for varargs. They use as many common components as possible to provide a fair comparison, and they have all been highly optimized. Some

highlights of the implementations include:

- Indirect function calls are handled as described by Pearce et al [66]. Function parameters are numbered contiguously starting immediately after their corresponding function variable, and when resolving indirect calls they are accessed as offsets to that function variable.

- Cycles are detected using Nuutila et al.'s [63] variant of Tarjan's algorithm, and they are collapsed using a union-find data structure with both union-by-rank and path compression heuristics.

- BLQ uses the incrementalization optimization described by Berndl et al. [7]. We use the BuDDy BDD library [54] to implement BDDs.

- LCD and HCD are both worklist algorithms—we use the worklist strategy LRF,[2] suggested by Pearce et al. [67], to prioritize the worklist. We also divide the worklist into two sections, *current* and *next*, as described by Nielson et al. [61]; items are selected from *current* and pushed onto *next*, and the two are swapped when *current* becomes empty. For our benchmarks, the divided worklist yields significantly better performance than a single worklist (the exact reason is unclear, other than the fact that the evaluation order of the nodes can significantly impact performance).

- Aside from BLQ, all the algorithms use sparse bitmaps to implement both the constraint graph and the points-to sets. The sparse bitmap implementation is taken from the GCC 4.1.1 compiler.

[2] **Least Recently Fired**—the node processed furthest back in time is given priority.

- We also experiment with the use of BDDs to represent the points-to sets. Unlike BLQ, which stores the entire points-to solution in a single BDD, we give each variable its own BDD to store its individual points-to set. For example, if $a \rightarrow \{b,c\}$ and $d \rightarrow \{c,e\}$, BLQ would have a single BDD representing the set of tuples $\{(a,b),(a,c),(d,c),(d,e)\}$. Instead, we give a a BDD representing the set $\{b,c\}$ and we give d a BDD representing the set $\{c,e\}$. The use of BDDs instead of sparse bitmaps is a simple modification that requires minimal changes to the code.

The benchmarks for our experiments are described in Table 2.2. Emacs is a text editor; Ghostscript is a postscript viewer; Gimp is an image manipulation program; Insight is a GUI overlaid on top of the gdb debugger; Wine is a Windows emulator; and Linux is the Linux operating system kernel. The constraint generator is separate from the constraint solvers: we generate constraints from the benchmarks using the CIL C front-end [60], ignoring any assignments involving types too small to hold a pointer. External library calls are summarized using hand-crafted function stubs. We pre-process the resulting constraint files using a variant of Offline Variable Substitution [70], which reduces the number of constraints by 60–77%. This pre-processing step takes less than a second for Emacs and Ghostscript, and between 1 and 3 seconds for Gimp, Insight, Wine, and Linux. The results reported are for these reduced constraint files; they include everything from reading in the constraint file from disk, creating the initial constraint graph, and solving that graph.

We run the experiments on a dual-core 1.83 GHz processor with 2 GB of memory, using the Ubuntu 6.10 Linux distribution. Though the processor is dual-core, the executables themselves are single-threaded. All executables are compiled using gcc-4.1.1 and the '– O3' optimization flag. We repeat each experiment three

	Emacs	Ghostscript	Gimp	Insight	Wine	Linux
HCD-Offline	0.05	0.17	0.26	0.23	0.51	0.62
HT	1.66	12.03	59.00	42.49	1,388.51	393.30
PKH	2.05	20.05	92.30	117.88	1,946.16	1,181.59
BLQ	4.74	121.60	167.56	265.94	5,117.64	5,144.29
LCD	3.07	15.23	39.50	39.02	1,157.10	327.65
HCD	0.46	49.55	59.70	73.92	**OOM**	659.74
HT+HCD	0.46	7.29	11.94	14.82	643.89	102.77
PKH+HCD	0.46	10.52	17.12	21.91	838.08	114.45
BLQ+HCD	5.81	115.00	173.46	257.05	4,211.71	4,581.91
LCD+HCD	0.56	7.99	12.50	15.97	492.40	86.74

Table 2.3: Performance (in seconds), using bitmaps for points-to sets. The HCD-Offline analysis is reported separately and not included in the times for those algorithms using HCD. The HCD algorithm runs out of memory (OOM) on the Wine benchmark.

times and report the smallest time; all the experiments have very low variance in performance.

2.3.3.1 Time and Memory Consumption

Table 2.3 shows the performance of the various algorithms. The times for HCD's offline analysis are shown separately and not included in the times for the various algorithms using HCD—they are small enough to be essentially negligible. Table 2.4 shows the memory consumption of the algorithms. Figure 2.3 graphically compares (using a log-scale) the performance of our combined algorithm LCD+HCD—the fastest of all the algorithms—against the current state-of-the-art algorithms. All these numbers were gathered using the sparse-bitmap implementations of the algorithms (except for BLQ).

BLQ's memory allocation is fairly constant across all the benchmarks. We

	Emacs	Ghostscript	Gimp	Insight	Wine	Linux
HT	17.7	84.9	279.0	231.5	1,867.2	901.3
PKH	17.6	83.9	269.5	194.7	1,448.3	840.7
BLQ	215.6	216.1	216.2	216.1	216.2	216.2
LCD	14.3	74.6	269.0	184.4	1,465.1	830.1
HCD	18.1	138.7	416.1	290.5	**OOM**	1,301.5
HT+HCD	12.4	80.8	253.9	186.5	1,391.4	842.5
PKH+HCD	13.9	79.1	264.6	186.0	1,430.2	807.5
BLQ+HCD	215.8	216.2	216.2	216.2	216.2	216.2
LCD+HCD	13.9	73.5	263.9	183.6	1,406.4	807.9

Table 2.4: Memory consumption (in megabytes), using bitmaps for points-to sets..

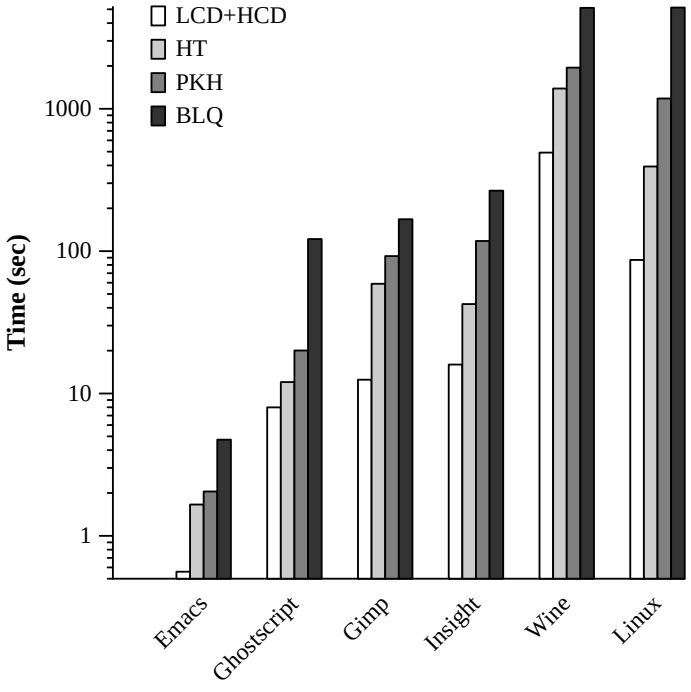


Figure 2.3: Performance (in seconds) of our new combined algorithm (LCD+HCD) versus three state-of-the art inclusion-based algorithms. Note that the Y-axis is log-scale.

allocate an initial pool of memory for the BDDs, which dominates the memory usage and is independent of benchmark size. While we can decrease the initial pool size for the smaller benchmarks without decreasing performance, there is no easy way to calculate the minimum pool size for a specific benchmark, so for all the benchmarks we use the smallest pool size that doesn't impair the performance of our largest benchmark.

It is interesting to note the vast difference in analysis time between Wine and Linux for all algorithms other than BLQ. While Wine has 32.5K fewer constraints than Linux, it takes 1.7–7.3× longer to be analyzed, depending on the algorithm used. This discrepancy points out the danger in using the size of the initial input to predict performance when other factors can have at least as much impact. Wine is a case in point: while its initial constraint graph is smaller than that of Linux, its final constraint graph at the end of the analysis is an order-of-magnitude larger than that of Linux, due mostly to Wine's larger average points-to set size. BLQ doesn't display this same behavior, because of its radically different analysis mechanism that uses BDDs and because it lacks cycle detection.

Comparing HT, PKH, BLQ, LCD, and HCD. Figure 2.4 compares the performance of the main algorithms by normalizing the times for HT, PKH, BLQ, and HCD by that of LCD. Focusing on the current state-of-the-art algorithms, HT is clearly the fastest: 1.9× faster than PKH and 6.5× faster than BLQ. LCD is on average 1.05× faster than HT and uses 1.2× less memory. HCD runs out of memory for Wine, but excluding that benchmark it is on average 1.8× slower than HT and 1.9× faster than PKH, and uses 1.4× more memory than HT.

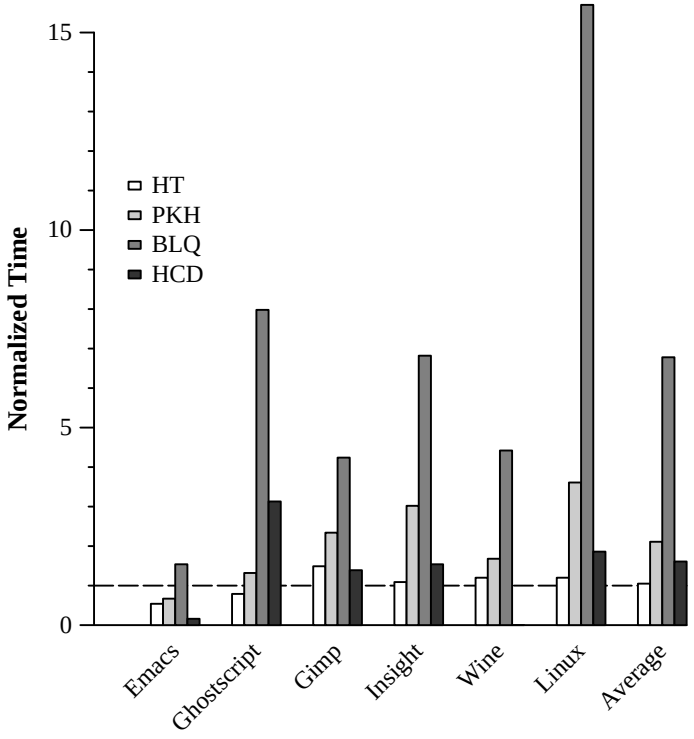


Figure 2.4: Performance comparison of individual benchmarks, where performance is normalized against LCD. HCD runs out of memory for Wine, so there is no HCD bar for that benchmark.

Effects of HCD. Figure 2.5 normalizes the performance of the main algorithms by that of their HCD-enhanced counterparts. On average, the use of HCD increases HT performance by 3.2×, PKH performance by 5×, BLQ performance by 1.1×, and LCD performance by 3.2×. HCD also leads to a small decrease in memory consumption for all the algorithms except BLQ—it decreases memory consumption by 1.2× for HT, by 1.1× for PKH, and by 1.02× for LCD. Most of the memory used by these algorithms comes from the representation of points-to sets. HCD improves performance by finding and collapsing cycles much earlier than normal, but it doesn't actually find many more cycles than were already detected without using HCD, so it doesn't significantly reduce the number of points-to sets that need to be maintained. HCD doesn't improve BLQ's performance by much because even though no extra effort is required to find cycles, there is still some overhead involved in collapsing those cycles. Also, the performance of BLQ depends on the sizes of the BDD representations of the constraint and points-to graphs, and because of the properties of BDDs, removing edges from the constraint graph can potentially increase the size of the constraint graph BDD.

The combination of our two new algorithms, LCD+HCD, yields the fastest algorithm among all those studied: It is 3.2× faster than HT, 6.4× faster than PKH, and 20.6× faster than BLQ.

2.3.3.2 Understanding the Results

There are a number of factors that determine the relative performance of these algorithms, but three of the most important are: (1) the number of nodes collapsed due to strongly-connected components; (2) the number of nodes searched during the depth-first traversals of the constraint graph; and (3) the number of propagations of points-to information across the edges of the constraint graph.

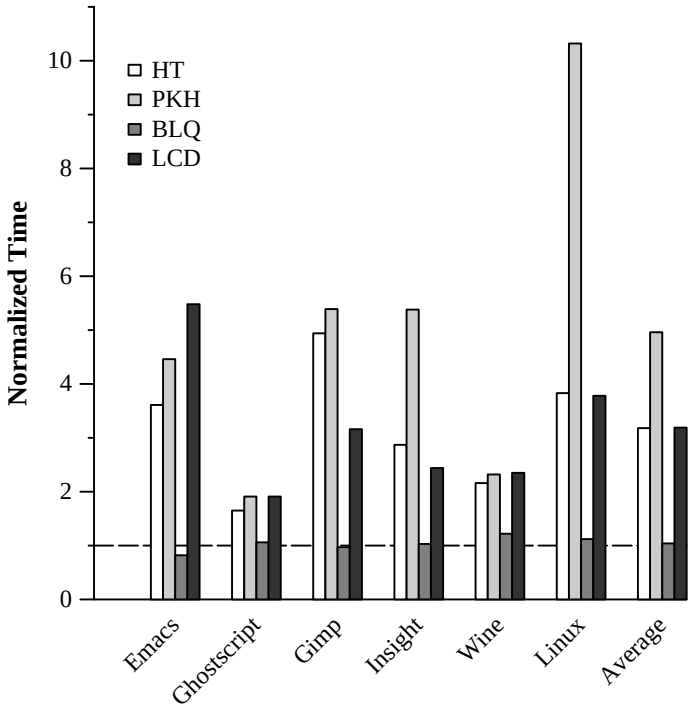


Figure 2.5: Performance comparison of the individual benchmarks, where the performance of each main algorithm is normalized against its respective HCD-enhanced counterpart.

	Emacs	**Ghostscript**	**Gimp**	**Insight**	**Wine**	**Linux**
HT	3.44	18.55	46.98	65.00	1,551.89	419.38
PKH	4.23	19.55	81.53	96.50	1,172.15	801.13
LCD	4.96	19.34	47.29	64.57	1,213.43	380.26
HCD	3.96	24.65	49.11	65.01	731.20	267.69
HT+HCD	2.58	15.65	33.69	42.33	737.37	209.90
PKH+HCD	3.06	14.70	33.71	43.20	744.35	172.43
LCD+HCD	3.09	13.69	33.04	43.17	625.82	183.97

Table 2.5: Performance (in seconds), using BDDs for points-to sets.

	Emacs	Ghostscript	Gimp	Insight	Wine	Linux
HT	33.1	49.3	100.7	100.0	811.2	274.3
PKH	33.2	33.6	50.4	66.8	226.4	182.1
LCD	33.2	33.2	40.1	33.9	251.1	73.5
HCD	33.1	37.1	36.8	37.0	239.6	65.8
HT+HCD	33.1	37.8	51.2	53.9	410.6	100.7
PKH+HCD	33.1	33.2	36.0	33.2	103.9	45.2
LCD+HCD	33.1	33.2	33.2	33.2	173.6	42.6

Table 2.6: Memory consumption (in megabytes), using BDDs for points-to sets.

The number of nodes collapsed is important because it reduces both the number of nodes and the number of edges in the constraint graph; the more nodes that are collapsed, the smaller the input and the more efficient the algorithm.

The depth-first searches are pure overhead due to cycle detection. As long as roughly as many cycles are being detected, then the fewer nodes that are searched the better.

The number of points-to information propagations is an important metric because propagation is one of the most expensive operations in the analysis. It is strongly influenced by both the number of cycles collapsed and by how quickly they are collapsed. If a cycle is not detected quickly, then points-to information could be redundantly circulated around the cycle a number of times.

We now examine these three quantities to help explain the performance results seen in the previous section. Due to its radically different analysis mechanism, we don't include BLQ in this examination.[3]

[3]It is difficult to find statistics to directly explain BLQ's performance relative to HT, PKH, LCD, and HCD. It doesn't use cycle detection, so it adds orders of magnitude more edges to the constraint graph—but propagation of points-to information is done simultaneously across all the edges using BDD operations, and the performance of the algorithm is due more to how well the BDDs compress

Nodes Collapsed. PKH is the only algorithm guaranteed to detect all strongly-connected components in the constraint graph; however, HT and LCD both do a very good job of finding and collapsing cycles—for each benchmark they detect and collapse over 99% of the nodes collapsed by PKH. HCD by itself doesn't do as well, collapsing only 46–74% of the nodes collapsed by PKH. This deficiency is primarily responsible for HCD's greater memory consumption.

Nodes Searched. HCD is, of course, the most efficient algorithm in terms of searching the constraint graph, since it doesn't search at all. HT is the next most efficient algorithm, because it only searches the subset of the graph necessary for resolving indirect constraints. PKH searches 2.6× as many nodes as HT, as it periodically searches the entire graph for cycles. LCD is the least efficient, searching 8× as many nodes as HT.

Propagations. LCD has the fewest propagations, showing that its greater effort at searching for cycles pays off by finding those cycles earlier than HT or PKH. HT has 1.8× as many propagations, and PKH has 2.2× as many. Since they both find as many cycles as LCD (as shown by the number of nodes collapsed), this difference is due to the relative amount of time it takes for each of the algorithms to find cycles. HCD has the most propagations, 5.2× as many as LCD. HCD finds cycles as soon as they are formed, so it finds them much faster than LCD does, but as shown above, it finds substantially fewer cycles than the other algorithms.

Effects of HCD. The main benefit of combining HCD with the other algorithms is that it helps these algorithms find cycles much sooner than they would on their

---

the constraint and points-to graphs than anything else.

own. While it does little to increase the number of nodes collapsed or decrease the number of nodes searched, it greatly decreases the number of propagations, because cycles are collapsed before the points-to information has a chance to propagate around the cycles. The addition of HCD decreases the number of propagations by 10× for HT and by 7.4× for both PKH and LCD.

Discussion. Despite its lazy nature, LCD searches more nodes than either HT or PKH, and it propagates less points-to information than either as well. It appears that being more aggressive pays off, which naturally leads to the question: could we do better by being even more aggressive? However, past experience has shown that we must carefully balance the work we do—too much aggression can lead to overhead that overwhelms any benefits it may provide. This point is shown in both Faehndrich et al.'s algorithm [29] and Pearce et al.'s original algorithm [67]. Both of these algorithms are very aggressive in seeking out cycles, and both are an order of magnitude slower than any of the algorithms evaluated in this paper.

2.3.3.3 Representing Points-to Sets

Table 2.4 shows that the memory consumption of all the algorithms that use sparse bitmaps is extremely high. Profiling reveals that the majority of this memory usage comes from the bitmap representation of points-to sets. BLQ, on the other hand, uses relatively little memory even for the largest benchmarks, due to its use of BDDs. It is thus natural to wonder how the other algorithms would compare—in terms of both analysis time and memory consumption—if they were to instead use BDDs to represent points-to sets.

Tables 2.5 and 2.6 show the performance and memory consumption of the modified algorithms. Figure 2.6 graphically shows the performance cost of the

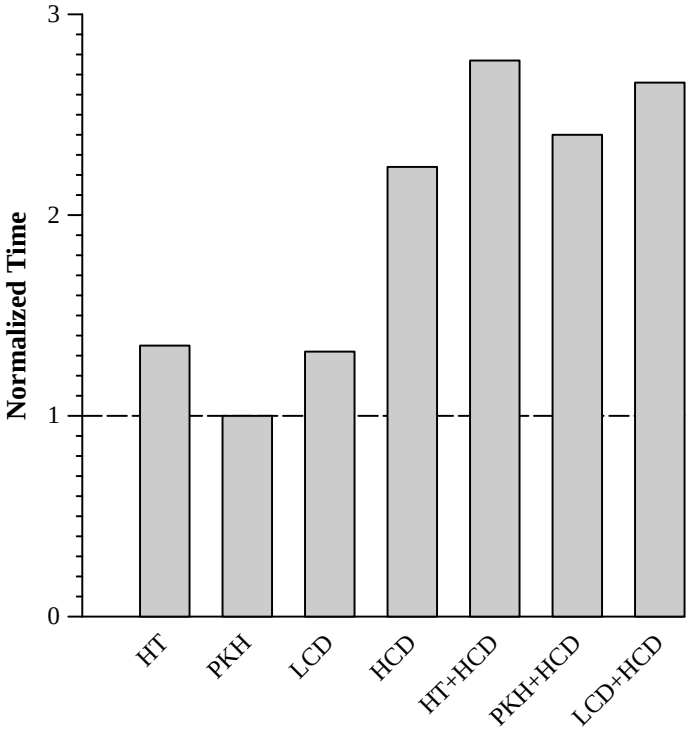


Figure 2.6: Performances of the BDD-based implementations normalized by their bitmap-based counterparts, averaged over all the benchmarks.

modified algorithms by normalizing them by their bitmap-based counterparts, and Figure 2.7 shows the memory savings by normalizing the bitmap-based algorithms by their BDD-based counterparts. As with BLQ, we allocate an initial pool of memory for the BDDs that is independent of the benchmark size, which is why memory consumption actually increases for the smallest benchmark, Emacs, and never goes lower than 33.1MB for any benchmark.

On average, the use of BDDs increases running time by $2\times$ while it decreases memory usage by $5.5\times$. Most of the extra time comes from a single function, *bdd_allsat*, which is used to extract all the elements of a set contained in a given BDD. This function is used when iterating through a variable's points-to set

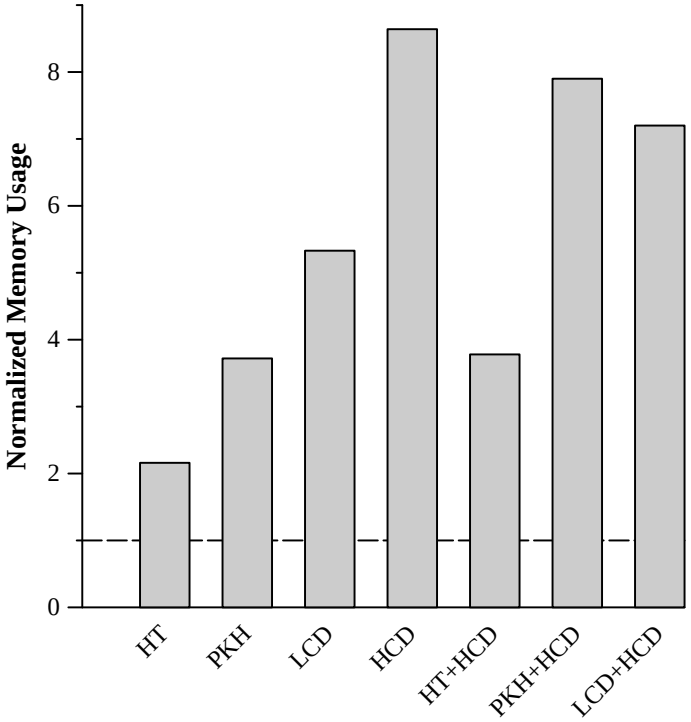


Figure 2.7: Memory consumption of the bitmap-based implementations normalized by their BDD-based counterparts, averaged over all the benchmarks.

while adding new edges according to the indirect constraints. However, both PKH and HCD are actually faster with BDDs on all benchmarks except for Emacs (Figure 2.6 shows that they are slower on average, but this is solely because of Emacs). These are the two algorithms that propagate the most points-to information across constraint edges. BDDs make this operation much faster than using sparse bitmaps, and this advantage makes up for the extra time taken by bdd_allsat.

When BDDs are used, HCD is less effective in improving performance than it was when using bitmaps because HCD decreases the number of propagations required, but using BDDs already makes propagation a fairly cheap operation. However, with BDDs, HCD's effect on memory consumption is much more noticeable, since the constraint graph represents a much larger proportion of the memory usage.

A possible optimization for the BDD-based points-to sets would be to cache the results of the *bdd_allsat* function, using a map from BDDs to a list of the BDD elements. While such a map seems to counter the memory-consumption advantage of using BDDs, the fact that many of the BDDs encode identical points-to sets would probably mitigate this problem.

2.4 Offline Optimizations

Offline optimizations are performed on the set of inclusion constraints prior to the actual inclusion-based analysis in order to reduce the input size of the problem. Rountev et al.'s Offline Variable Substitution (OVS) is an example of this technique. Prior work (including OVS) targets *pointer equivalence,* i.e., detecting and collapsing variables that are guaranteed to have identical points-to sets. Section 2.4.1 describes several techniques that also target pointer equivalence and improve on the state-of-the-art. Section 2.4.2 describes an optimization that tar-

gets *location equivalence*, a new type of equivalence that has never been defined or exploited for pointer analysis prior to this work.

2.4.1 Pointer Equivalence

Let $\mathcal{V}$ be the set of all program variables and $\mathcal{N}$ be the set of natural numbers; for $v \in \mathcal{V}$: $pts(v) \subseteq \mathcal{V}$ is v's final points-to set, and $pe(v) \in \mathcal{N}$ is the *pointer equivalence label* of v. Variables x and y are pointer equivalent iff $pts(x) = pts(y)$. Our goal is to assign pointer equivalence labels such that $pe(x) = pe(y)$ implies that x and y are pointer equivalent. Pointer equivalent variables can safely be collapsed together in the constraint graph to reduce both the number of nodes and edges in the graph. The benefits are two-fold: (1) there are fewer points-to sets to maintain; and (2) there are fewer propagations of points-to information along the edges of the constraint graph.

The offline optimization begins by using the set of inclusion constraints to create an *offline constraint graph*,[4] with VAR nodes to represent each variable, REF nodes to represent each dereferenced variable, and ADR nodes to represent each address-taken variable. A REF node $*a$ stands for the unknown points-to set of variable a, while ADR node $\&a$ stands for the address of variable a. Edges represent the inclusion relationships: $a \supseteq \{b\}$ yields edge $\&b \rightarrow a$; $a \supseteq b$ yields $b \rightarrow a$; $a \supseteq *b$ yields $*b \rightarrow a$; and $*a \supseteq b$ yields $b \rightarrow *a$.

Before describing the optimizations, we first explain the concepts of *direct* and *indirect* nodes [70]. Direct nodes have all of their points-to relations explicitly represented in the constraint graph: for direct node x and the set of nodes $\mathcal{S} = \{y : y \rightarrow x\}$, $pts(x) = \bigcup_{y \in \mathcal{S}} pts(y)$. Indirect nodes are those that may have points-to

[4]The offline constraint graph is akin to the *subset graph* described by Rountev et al. [70].

relations that are not represented in the constraint graph. All REF nodes are indirect because the unknown variables that they represent may have their own points-to relations. VAR nodes are indirect if they (1) have had their address taken, which means that they can be referenced indirectly via a REF node; (2) are the formal parameter of an function targeted by an indirect call; or (3) are assigned the return value of an indirect function call. All other VAR nodes are direct.

All indirect nodes are conservatively treated as possible sources of points-to information, and therefore each is given a distinct pointer equivalence label at the beginning of the algorithm. ADR nodes are definite sources of points-to information, and they are also given distinct labels. For convenience, we will use the term 'indirect node' to refer to both ADR nodes and true indirect nodes because our optimizations treat them equivalently.

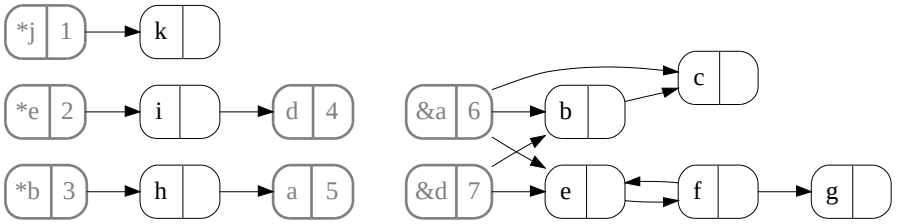
Figure 2.8 shows a set of constraints and the corresponding offline constraint graph. In Figure 2.8 all the REF and ADR nodes are marked indirect, as well as VAR nodes *a* and *d*, because they have their address taken. Because *a* and *d* can now be accessed indirectly through pointer dereference, we can no longer assume that they only acquire points-to information via nodes *h* and *i*, respectively.

2.4.1.1 Hash-based Value Numbering (HVN)

The goal of HVN is to give each direct node a pointer equivalence label such that two nodes share the same label only if they are pointer equivalent. HVN can also identify non-pointers—variables that are guaranteed to never point to anything. Non-pointers can arise in languages with weak types systems, such as C: the constraint generator can't rely on the variables' type declarations to determine whether a variable is a pointer or not, so it conservatively assumes that everything is a pointer. HVN can eliminate many of these superfluous variables; they are iden-

$b \supseteq \{a\}$	$a \supseteq h$	$h \supseteq *b$
$b \supseteq \{d\}$	$c \supseteq b$	$i \supseteq *e$
$c \supseteq \{a\}$	$d \supseteq i$	$k \supseteq *j$
$e \supseteq \{a\}$	$e \supseteq f$	
$e \supseteq \{d\}$	$f \supseteq e$	
	$g \supseteq f$	

(a) Set of constraints.



(b) Offline constraint graph.

Figure 2.8: Example offline constraint graph. Indirect nodes are grey and have already been given their pointer equivalence labels. Direct nodes are black and have not been given pointer equivalence labels.

tified by assigning a pointer equivalence label of 0. The algorithm proceeds as follows:

1. Find and collapse strongly-connected components (SCCs) in the offline constraint graph. If any node in the SCC is indirect, the entire SCC is indirect. In Figure 2.8, e and f are collapsed into a single (direct) node.

2. Proceeding in topological order, for each direct node x let $\mathcal{L}$ be the set of positive incoming pointer equivalence labels, i.e., $\mathcal{L} = \{pe(y) : y \rightarrow x \wedge pe(y) \neq 0\}$. There are three cases:

   (a) $\mathcal{L}$ is empty. Then x is a non-pointer and $pe(x) = 0$.

   Explanation: in order for x to potentially be a pointer, there must exist a path to x either from an ADR node or some indirect node. If there is

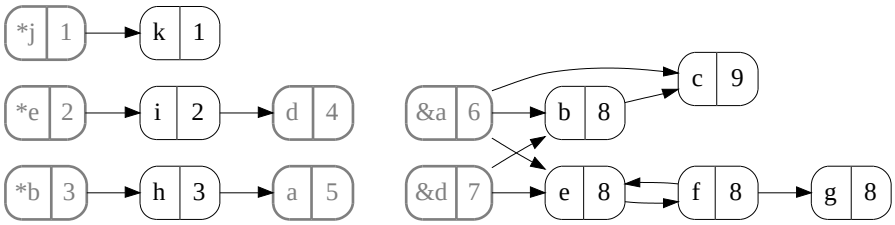


Figure 2.9: The assignment of pointer equivalence labels after HVN.

no such path, then x must be a non-pointer.

(b) $\mathcal{L}$ is a singleton, with $p \in \mathcal{L}$. Then $pe(x) = p$.

Explanation: if every points-to set coming in to x is identical, then x's points-to set, being the union of all the incoming points-to sets, must be identical to the incoming sets.

(c) $\mathcal{L}$ contains multiple labels. The algorithm looks up $\mathcal{L}$ in a hashtable to see if it has encountered the set before. If so, it assigns $pe(x)$ the same label; otherwise it creates a new label, stores it in the hashtable, and assigns it to $pe(x)$.

Explanation: x's points-to set is the union of all the incoming points-to sets; x must be equivalent to any node whose points-to set results from unioning the same incoming points-to sets.

Following these steps for Figure 2.8, the final assignment of pointer equivalence labels for the direct nodes is shown in Figure 2.9. Once we have assigned pointer equivalence labels, we merge nodes with identical labels and eliminate all edges incident to nodes labeled 0.

Complexity. The complexity of HVN is linear in the size of the graph. Using Tarjan's algorithm for detecting SCCs [79], step 1 is linear. The algorithm then

visits each direct node exactly once and examines its incoming edges. This step is also linear.

Comparison to OVS. HVN is similar to Rountev et al.'s [70] OVS optimization. The main difference lies in our insight that labeling the condensed offline constraint graph is essentially equivalent to performing value-numbering on a block of straight-line code, and therefore we can adapt the classic compiler optimization of hash-based value numbering for this purpose. The advantage lies in step 2*c*: in this case OVS would give the direct node a new label without checking to see if any other direct nodes have a similar set of incoming labels, potentially missing a pointer equivalence. In the example, OVS would not discover that *b* and *e* are equivalent and would give them different labels.

2.4.1.2 Extending HVN

HVN does not find all pointer equivalences that can be detected prior to pointer analysis because it does not interpret the *union* and *dereference* operators. Recall that the union operator is implicit in the offline constraint graph: for direct node x with incoming edges from nodes y and z, $pts(x) = pts(y) \cup pts(z)$. By interpreting these operators, we can increase the number of pointer equivalences detected, at the cost of additional time and space.

HR algorithm. By interpreting the dereference operator, we can relate a VAR node v to its corresponding REF node $*v$. There are two relations of interest:

1. $\forall x,y \in \mathcal{V} : pe(x) = pe(y) \Rightarrow pe(*x) = pe(*y)$.
2. $\forall x \in \mathcal{V} : pe(x) = 0 \Rightarrow pe(*x) = 0$.

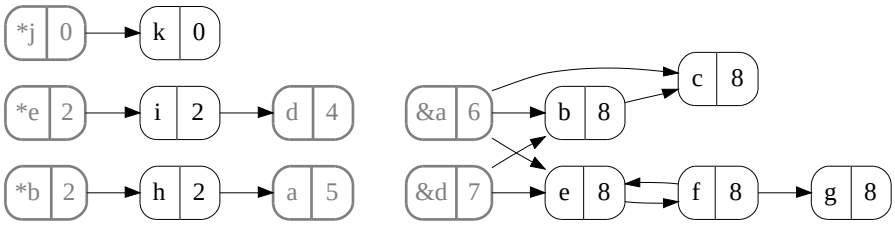


Figure 2.10: The assignment of pointer equivalence labels after HR and HU.

The first relation states that if variables x and y are pointer-equivalent, then so are $*x$ and $*y$. If x and y are pointer-equivalent, then by definition $*x$ and $*y$ will be identical. Whereas HVN would give them unique pointer equivalence labels, we can now assign them the same label. By doing so, we may find additional pointer equivalences that had previously been hidden by the different labels.

The second relation states that if variable x is a non-pointer, then $*x$ is also a non-pointer. It may seem odd to have a constraint that dereferences a non-pointer, but this can happen when code that initializes pointer values is linked but never called, for example with library code. Exposing this relationship can help identify additional non-pointers and pointer equivalences.

Figure 2.10 provides an example. HVN assigns b and e identical labels; the first relation above tells us we can assign $*b$ and $*e$ identical labels, which exposes the fact that i and h are equivalent to each other, which HVN missed. Also, variable j is not mentioned in the constraints, and therefore the VAR node j isn't shown in the graph, and it is assigned a pointer equivalence label of 0. The second relation above tells us that because $pe(j) = 0$, $pe(*j)$ should also be 0; therefore both $*j$ and k are non-pointers and can be eliminated.

The simplest method for interpreting the dereference operator is to iteratively apply HVN to its own output until it converges to a fixed point. Each iteration

collapses equivalent variables and eliminates non-pointers, fulfilling the two relations we describe. This method adds an additional factor of $O(n)$ to the complexity of the algorithm, since in the worst case it eliminates a single variable in each iteration until there is only one variable left. The complexity of HR is therefore $O(n^2)$, but in practice we observe that this method generally exhibits linear behavior.

HU algorithm. By interpreting the union operator, we can more precisely track the relations among points-to sets. Figure 2.10 gives an example in VAR node c. Two different pointer equivalence labels reach c, one from $\&a$ and one from b. HVN therefore gives c a new pointer equivalence label. However, $pts(b) \supseteq pts(\&a)$, so when they are unioned together the result is simply $pts(b)$. By keeping track of this fact, we can assign c the same pointer equivalence label as b.

Let f_n be a fresh number unique to n; the algorithm will use these fresh values to represent unknown points-to information. The algorithm operates on the condensed offline constraint graph as follows:

1. Initialize points-to sets for each node. $\forall v \in \mathcal{V}: \; pts(\&v) = \{v\}; \; pts(*v) = \{f_{*v}\}$; if v is direct then $pts(v) = \emptyset$, else $pts(v) = \{f_v\}$.

2. In topological order: for each node x, let $\mathcal{S} = \{y : y \rightarrow x\} \cup \{x\}$. Then $pts(x) = \bigcup_{y \in \mathcal{S}} pts(y)$.

3. Assign labels s.t. $\forall x, y \in V : pts(x) = pts(y) \Leftrightarrow pe(x) = pe(y)$.

Since this algorithm is effectively computing the transitive closure of the constraint graph, it has a complexity of $O(n^3)$. While this is the same complexity as the pointer analysis itself, HU is significantly faster because, unlike the pointer

analysis, we do not add additional edges to the offline constraint graph, making the offline graph much smaller than the graph used by the pointer analysis.

Putting It Together: HRU. The HRU algorithm combines the HR and HU algorithms to interpret both the dereference and union operators. HRU modifies HR to iteratively apply the HU algorithm to its own output until it converges to a fixed point. Since the HU algorithm is $O(n^3)$ and HR adds a factor of $O(n)$, HRU has a complexity of $O(n^4)$. As with HR this worst-case complexity is not observed in practice; however it is advisable to first apply HVN to the original constraints, then apply HRU to the resulting set of constraints. HVN significantly decreases the size of the offline constraint graph, which decreases both the time and memory consumption of HRU.

2.4.2 Location Equivalence

Let $\mathcal{V}$ be the set of all program variables and $\mathcal{N}$ be the set of natural numbers; for $v \in \mathcal{V}$: $pts(v) \subseteq \mathcal{V}$ is v's points-to set, and $le(v) \in \mathcal{N}$ is the *location equivalence label* of v. Variables x and y are location equivalent iff $\forall z \in \mathcal{V} : x \in pts(z) \Leftrightarrow y \in pts(z)$. Our goal is to assign location equivalence labels such that $le(x) = le(y)$ implies that x and y are location equivalent. Location equivalent variables can safely be collapsed together in all points-to sets, providing two benefits: (1) the points-to sets consume less memory; and (2) since the points-to sets are smaller, points-to information is propagated more efficiently across the edges of the constraint graph.

Without any pointer information it is impossible to compute all location equivalences, as can be seen by the definition of location equivalence given above. However, since points-to sets are never split during the pointer analysis, any vari-

ables that are location equivalent at the beginning of the analysis are guaranteed to be location equivalent at the end. We can therefore safely compute a subset of the equivalences prior to the pointer analysis. We use the same offline constraint graph as we use to find pointer equivalence, but we will be labeling ADR nodes instead of direct nodes. The algorithm assigns each ADR node a label based on its outgoing edges such that two ADR nodes have the same label iff they have the same set of outgoing edges. In other words, ADR nodes &a and &b are assigned the same label iff, in the constraints, $\forall z \in \mathcal{V} : z \supseteq \{a\} \Leftrightarrow z \supseteq \{b\}$. In Figure 2.8, the ADR nodes &a and &d would be assigned the same location equivalence label.

While location and pointer equivalences can be computed independently, it is more precise to compute location equivalence *after* we have computed pointer equivalence. We modify the criterion to require that ADR nodes &a and &b are assigned the same label iff $\forall y,z \in V, (y \supseteq \{a\} \wedge z \supseteq \{b\}) \Rightarrow pe(y) = pe(z)$. In other words, we don't require that the two ADR nodes have the same set of outgoing edges, but rather that the nodes incident to the ADR nodes have the same set of pointer equivalence labels.

Once the algorithm assigns location equivalence labels, it merges all ADR nodes that have identical labels. These merged ADR nodes are each given a fresh name. Points-to set elements will come from this new set of fresh names rather than from the original names of the merged ADR nodes, thereby saving space, since a single fresh name corresponds to multiple ADR nodes. However, we must make a simple change to the subsequent pointer analysis to accommodate this new naming scheme. When adding new edges from indirect constraints, the pointer analysis must translate from the fresh names in the points-to sets to the original names corresponding to the VAR nodes in the constraint graph. To facilitate this translation the analysis creates a one-to-many mapping between the fresh names and the orig-

inal ADR nodes that it merged together. In Figure 2.8, since ADR nodes &a and &d are given the same location equivalence label, they will be merged together and assigned a fresh name such as &l. Any points-to sets that formerly would have contained a and d will instead contain l; when adding additional edges from an indirect constraint that references l, the pointer analysis will translate l back to a and d to correctly place the edges in the online constraint graph.

Complexity. LE is linear in the size of the constraint graph. The algorithm scans through the constraints, and for each constraint $a \supseteq \{b\}$ it inserts $pe(a)$ into ADR node &b's set of pointer equivalence labels. This step is linear in the number of constraints (i.e., graph edges). It then visits each ADR node, and it uses a hash table to map from that node's set of pointer equivalence labels to a single location equivalence label. This step is also linear.

2.4.3 Evaluation

Using a suite of six open-source C programs, which range in size from 169K to 2.17M LOC, we compare the analysis times and memory consumption of OVS, HVN, HRU, and HRU+LE (HRU coupled with LE). We then use three different state-of-the-art inclusion-based pointer analyses—Pearce et al. [66] (PKH), Heintze and Tardieu [41] (HT), and Hardekopf and Lin [37] (HL)—to compare the optimizations' effects on the pointer analyses' analysis time and memory consumption. These pointer analyses are all field-insensitive and implemented in a common framework, re-using as much code as possible to provide a fair comparison.

The offline optimizations and the pointer analyses are written in C++ and handle all aspects of the C language except for varargs. We use sparse bitmaps taken from GCC 4.1.1 to represent the constraint graph and points-to sets. The

Name	Description	LOC	Constraints
Emacs-21.4a	text editor	169K	83,213
Ghostscript-8.15	postscript viewer	242K	169,312
Gimp-2.2.8	image manipulation	554K	411,783
Insight-6.5	graphical debugger	603K	243,404
Wine-0.9.21	windows emulator	1,338K	713,065
Linux-2.4.26	linux kernel	2,172K	574,788

Table 2.7: Benchmarks: For each benchmark we show the number of lines of code (computed as the number of non-blank, non-comment lines in the source files), a description of the benchmark, and the number of constraints generated by the CIL front-end.

constraint generator is separate from the constraint solvers; we generate constraints from the benchmarks using the CIL C front-end [60], ignoring any assignments involving types too small to hold a pointer. External library calls are summarized using hand-crafted function stubs.

The benchmarks for our experiments are described in Table 2.7. We run the experiments on an Intel Core Duo 1.83 GHz processor with 2 GB of memory, using the Ubuntu 6.10 Linux distribution. Though the processor is dual-core, the executables themselves are single-threaded. All executables are compiled with GCC 4.1.1 and the '–O3' optimization flag. We repeat each experiment three times and report the smallest time; all the experiments have very low variance in performance. Times include everything from reading the constraint file from disk to computing the final solution.

2.4.3.1 Cost of Optimizations

Tables 2.8 and 2.9 show the analysis time and memory consumption, respectively, of the offline optimizations on the six benchmarks. OVS and HVN have roughly the same times, with HVN using 1.17$\times$ more memory than OVS. On aver-

	Emacs	Ghostscript	Gimp	Insight	Wine	Linux
OVS	0.29	0.60	1.74	0.96	3.57	2.34
HVN	0.29	0.61	1.66	0.95	3.39	2.36
HRU	0.49	2.29	4.31	4.28	9.46	7.70
HRU+LE	0.53	2.54	4.75	4.64	10.41	8.47

Table 2.8: Offline analysis times (sec).

	Emacs	Ghostscript	Gimp	Insight	Wine	Linux
OVS	13.1	28.1	61.1	39.1	110.4	96.2
HVN	14.8	32.5	71.5	44.7	134.8	114.8
HRU	14.8	32.5	71.5	44.7	134.8	114.8
HRU+LE	14.8	32.5	71.5	44.7	134.8	114.8

Table 2.9: Offline analysis memory (MB).

age, HRU and HRU+LE are 3.1× slower and 3.4× slower than OVS, respectively. Both HRU and HRU+LE have the same memory consumption as HVN. As stated earlier, these algorithms are run on the output of HVN in order to improve analysis time and conserve memory; their times are the sum of their running time and the HVN running time, while their memory consumption is the maximum of their memory usage and the HVN memory usage. In all cases, the HVN memory usage is greater.

Figure 2.11 shows the effect of each optimization on the number of constraints for each benchmark. On average OVS reduces the number of constraints by 63.4%, HVN by 69.4%, HRU by 77.4%, and HRU+LE by 79.9%. HRU+LE, our most aggressive optimization, takes 3.4× longer than OVS, while it only reduces the number of constraints by an additional 16.5%. However, inclusion-based analysis is $O(n^3)$ time and $O(n^2)$ space, so even a relatively small reduction in the input size can have a significant effect, as we'll see in the next section.

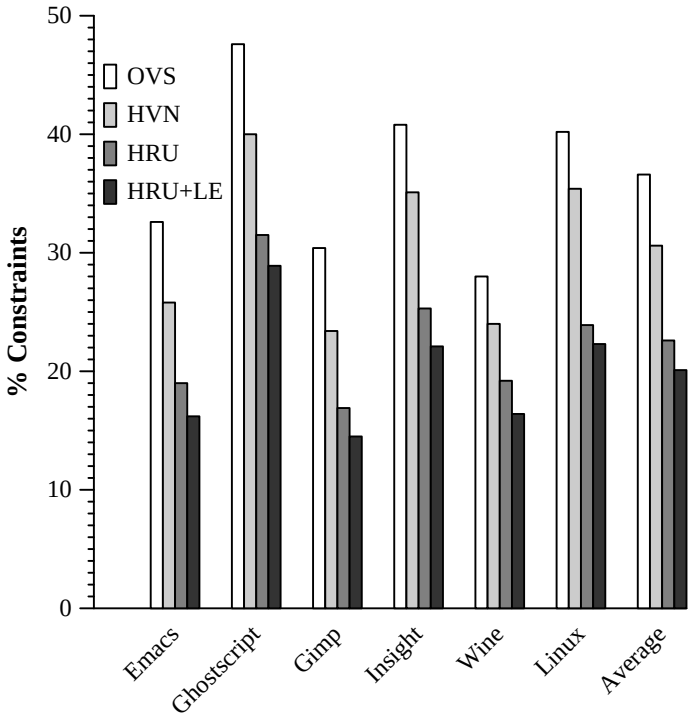


Figure 2.11: Percent of the original number of constraints that is generated by each optimization.

	Emacs	Ghostscript	Gimp	Insight	Wine	Linux
OVS	1.99	19.15	99.22	121.53	1,980.04	1,202.78
HVN	1.60	17.08	87.03	111.81	1,793.17	1,126.90
HRU	0.74	13.31	38.54	57.94	1,072.18	598.01
HRU+LE	0.74	9.50	21.03	33.72	731.49	410.23

Table 2.10: Online analysis times for the PKH algorithm (sec).

	Emacs	Ghostscript	Gimp	Insight	Wine	Linux
OVS	1.63	13.58	64.45	46.32	**OOM**	410.52
HVN	1.84	12.84	59.68	42.70	**OOM**	393.00
HRU	0.70	9.95	37.27	37.03	1,087.84	464.51
HRU+LE	0.54	8.82	18.71	23.35	656.65	332.36

Table 2.11: Online analysis times for the HT algorithm (sec).

2.4.3.2 Benefit of Optimizations

Tables 2.10–2.15 give the analysis times and memory consumption for three pointer analyses—PKH, HT, and HL—as run on the results of each offline optimization; OOM indicates the analysis ran out of memory. The data is summarized in Figure 2.12, which gives the average performance and memory improvement for the three pointer analyses for each offline algorithm as compared to OVS. The offline analysis times are added to the pointer analysis times to make the overall analysis time comparison.

	Emacs	Ghostscript	Gimp	Insight	Wine	Linux
OVS	1.07	9.15	17.55	20.45	534.81	103.37
HVN	0.68	8.14	13.69	17.23	525.31	91.76
HRU	0.32	7.25	10.04	12.70	457.49	75.21
HRU+LE	0.51	6.67	8.39	13.71	345.56	79.99

Table 2.12: Online analysis times for the HL algorithm (sec).

	Emacs	Ghostscript	Gimp	Insight	Wine	Linux
OVS	23.1	102.7	418.1	251.4	1,779.7	1,016.5
HVN	17.7	83.9	269.5	194.8	1,448.5	840.8
HRU	12.8	68.0	171.6	165.4	1,193.7	590.4
HRU+LE	6.9	23.8	56.1	58.6	295.9	212.4

Table 2.13: Online analysis memory for the PKH algorithm (MB).

	Emacs	Ghostscript	Gimp	Insight	Wine	Linux
OVS	22.5	97.2	359.7	266.9	**OOM**	1,006.8
HVN	17.7	85.0	279.0	231.5	**OOM**	901.3
HRU	10.8	70.3	205.3	156.7	1,533.0	700.7
HRU+LE	6.4	34.9	86.0	69.4	820.9	372.2

Table 2.14: Online analysis memory for the HT algorithm (MB).

	Emacs	Ghostscript	Gimp	Insight	Wine	Linux
OVS	21.0	93.9	415.4	239.7	1,746.3	987.8
HVN	13.9	73.5	263.9	183.7	1,463.5	807.9
HRU	9.2	63.3	170.7	121.9	1,185.3	566.6
HRU+LE	4.5	22.2	33.4	27.6	333.1	162.6

Table 2.15: Online analysis memory for the HL algorithm (MB).

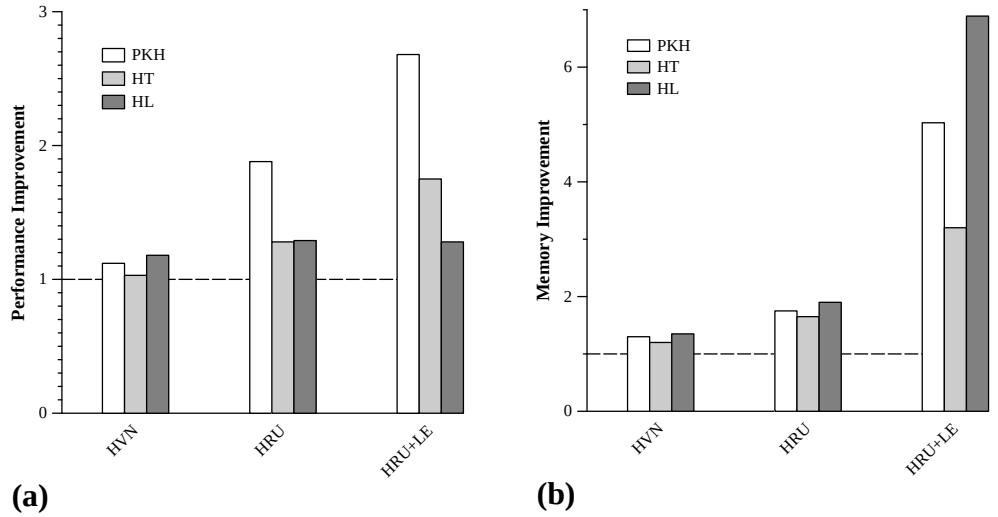


Figure 2.12: **(a)** Average performance improvement over OVS; **(b)** Average memory improvement over OVS. For each graph, and for each offline optimization $X \in$ {HVN, HRU, HRU+LE}, we compute $\frac{OVS_{time/memory}}{X_{time/memory}}$. Larger is better.

Analysis Time. For all three pointer analyses, HVN only moderately improves analysis time over OVS, by 1.03–1.18×. HRU has a greater effect despite its much higher offline analysis times; it improves analysis time by 1.28–1.88×. HRU+LE has the greatest effect; it improves analysis time by 1.28–2.68×. An important factor in the analysis time of these algorithms is the number of times they propagate points-to information across constraint edges. PKH is the least efficient of the algorithms in this respect, propagating much more information than the other two; hence it benefits more from the offline optimizations. HL propagates the least amount of information and therefore benefits the least.

Memory. For all three pointer analyses HVN only moderately improves memory consumption over OVS, by 1.2–1.35×. All the algorithms benefit significantly from HRU, using 1.65–1.90× less memory than for OVS. HRU's greater reduc-

tion in constraints makes for a smaller constraint graph and fewer points-to sets. HRU+LE has an even greater effect: HT uses 3.2× less memory, PKH uses 5× less memory, and HL uses almost 7× less memory. HRU+LE doesn't further reduce the constraint graph or the number of points-to sets, but on average it cuts the average points-to set size in half.

Room for Improvement. Despite aggressive offline optimization in the form of HRU plus the efforts of online cycle detection, there are still a significant number of pointer equivalences that we do not detect in the final constraint graph. The number of actual pointer equivalence classes is much smaller than the number of detected equivalence classes, by almost 4× on average. In other words, we could conceivably shrink the online constraint graph by almost 4× if we could do a better job of finding pointer equivalences. This is an interesting area for future work. On the other hand, we do detect a significant fraction of the actual location equivalences—we detect 90% of the actual location equivalences in the five largest benchmarks, though for the smallest (Emacs) we only detect 41%. Thus there is not much room to improve on the LE optimization.

2.4.3.3 Bitmaps vs. BDDs.

The data structure used to represent points-to sets for the pointer analysis can have a great effect on the analysis time and memory consumption of the analysis. Section 2.3.3 compares the use of sparse bitmaps versus BDDs to represent points-to sets and find that on average the BDD implementation is 2× slower but uses 5.5× less memory than the bitmap implementation. To make a similar comparison testing the effects of our optimizations, we implement two versions of each pointer analysis: one using sparse bitmaps to represent points-to sets, the other us-

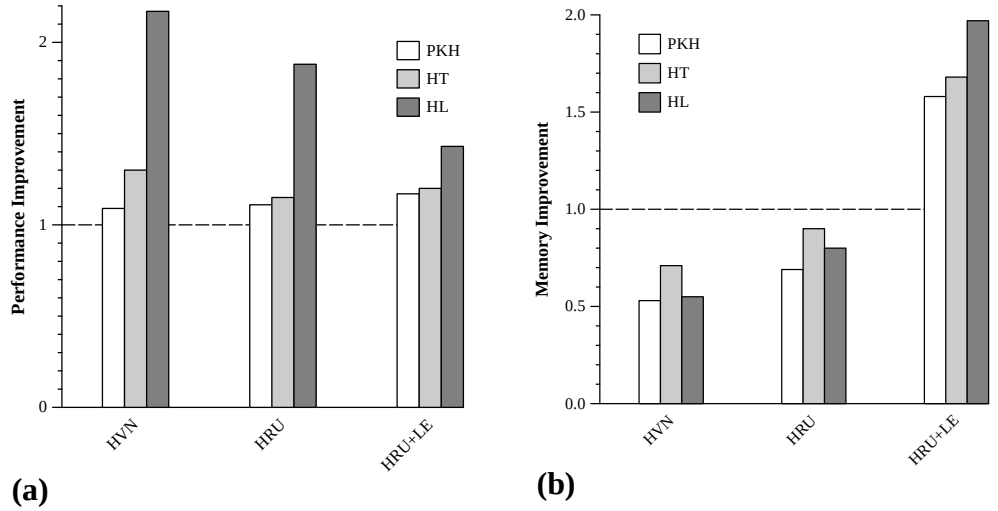


Figure 2.13: **(a)** Average performance improvement over BDDs;**(b)** Average memory improvement over BDDs. Let *BDD* be the BDD implementation and *BIT* be the bitmap implementation; for each graph we compute $\frac{BDD_{time/memory}}{BIT_{time/memory}}$. Larger is better.

ing BDDs for the same purpose. Unlike BDD-based pointer analyses [7, 81] which store the entire points-to solution in a single BDD, we give each variable its own BDD to store its individual points-to set. For example, if $v \rightarrow \{w,x\}$ and $y \rightarrow \{x,z\}$, the BDD-based analyses would have a single BDD that represents the set of tuples $\{(v,w),(v,x),(y,x),(y,z)\}$. Instead, we give v a BDD that represents the set $\{w,x\}$ and we give y a BDD that represents the set $\{w,z\}$. The two BDD representations take equivalent memory, but our representation is a simple modification that requires minimal changes to the existing code.

The results of our comparison are shown in Figure 2.13. We find that for HVN and HRU, the bitmap implementations on average are 1.4–1.5× faster than the BDD implementations but use 3.5–4.4× more memory. However, for HRU+LE the bitmap implementations are on average 1.3× faster and use 1.7× *less* memory than the BDD implementations, because the LE optimization significantly shrinks

the points-to sets of the variables.

2.5 Chapter Summary

In this chapter, we demonstrate how to greatly increase the scalability of inclusion-based pointer analysis, the most precise type of flow- and context-insensitive pointer analysis. Our strategy is to exploit both *pointer equivalence* (pointers with identical points-to sets) and *location equivalence* (variables pointed-to by identical sets of pointers).

A major technique for exploiting pointer equivalence is *cycle detection*, i.e., detecting and collapsing cycles in the constraint graph. This chapter describes two new techniques for cycle detection that outperform the previous state-of-the-art: *Lazy Cycle Detection* and *Hybrid Cycle Detection*. A second technique to exploit pointer equivalence is *offline variable substitution*. This chapter describes three new offline techniques for detecting pointer equivalent variables that detect more equivalences than the previous state-of-the-art: *HVN*, *HR*, and *HU* (and their combination, *HRU*).

While pointer equivalence has been previously exploited for optimizing inclusion-based analysis, location equivalence is a brand-new technique introduced here for the first time. This chapter describes an offline technique for conservatively detecting location equivalent variables prior to the inclusion-based analysis. This technique detects a majority of the equivalent variables in most benchmarks and greatly reduces the memory consumption of the analysis.

Chapter 3

Flow-Sensitive Analysis

Flow-sensitive pointer analysis has proven particularly difficult in terms of scalability; previous state-of-the-art flow-sensitive algorithms can analyze programs with only a few tens of thousands of lines of code. This chapter describes two new algorithms for flow-sensitive pointer analysis that increase scalability by two orders of magnitude, enabling the analysis of millions of lines of code in less than 15 minutes. Section 3.1 provides background on flow-sensitive pointer analysis, including why flow-sensitivity is important and some insight into why flow-sensitive analysis is so expensive. Section 3.2 discusses related work for scalable flow-sensitive analysis, then Sections 3.3 and 3.4 describe our new algorithms and evaluate their performance with respect to the current state-of-the-art. Portions of the work described in this chapter has been previously published by Hardekopf and Lin [39].

3.1 Background

This section briefly describes flow-sensitive pointer analysis and enumerates the major challenges in making the analysis practical for large programs. Further details on the basic flow-sensitive pointer analysis algorithm are described by Hind et al. [44].

3.1.1 Flow-Sensitive Pointer Analysis

Flow-sensitive pointer analysis respects a program's control flow and computes a separate solution for each program point, in contrast to a flow-insensitive analysis, which ignores statement ordering and computes a single solution that is conservatively correct for all program points.

Traditional flow-sensitive pointer analysis uses an iterative dataflow analysis framework, which employs a lattice of dataflow facts $\mathcal{L}$, a meet operator on the lattice, and a family of monotone transfer functions $f_i : \mathcal{L} \rightarrow \mathcal{L}$ that map lattice elements to other lattice elements. For pointer analysis the lattice elements are points-to graphs, the meet operator is set union, and each transfer function computes the effects of a program statement to transform an input points-to graph into an output points-to graph. Analysis is carried out on the *control-flow graph* (CFG), a directed graph $G = \langle N, E \rangle$ with a finite set of nodes (or *program points*), N, corresponding to program statements and a set of edges $E \subseteq N \times N$ corresponding to the control flow between statements. To ensure decidability of the analysis branch conditions are uninterpreted and branches are treated as non-deterministic.

Each node k of the CFG maintains two points-to graphs: IN_k, representing the incoming pointer information, and OUT_k, representing the outgoing pointer information. Each node is associated with a transfer function that transforms IN_k to OUT_k, characterized by the sets GEN_k and KILL_k, which represent the pointer information generated by the node and killed by the node, respectively. The contents of these two sets depend on the particular program statement associated with node k, and the contents can vary over the course of the analysis as new pointer information is accumulated (though the transfer function is still guaranteed to be monotonic). The analysis iteratively computes the following two functions for all nodes k until convergence:

$$\text{IN}_k = \bigcup_{x \in pred(k)} \text{OUT}_x \tag{3.1}$$

$$\text{OUT}_k = \text{GEN}_k \cup (\text{IN}_k - \text{KILL}_k) \tag{3.2}$$

The KILL set determines whether the analysis performs a *strong* or *weak* update to the left-hand side of an assignment. When the left-hand side definitely refers to a single memory location v, a strong update occurs in which the KILL set is used to remove all points-to relations $v \rightarrow x$ prior to updating v with a new set of points-to relations. If the left-hand side cannot be determined to point to a single memory location, then a weak update occurs: The analysis cannot be sure *which* of the possible memory locations should actually be updated by the assignment, so to be conservative it must set KILL to the empty set to preserve all of the existing points-to relations.

3.1.2 The Importance of Flow-Sensitive Pointer Analysis

Some previous work has created a perception that the extra precision of flow-sensitive pointer analysis is not beneficial [45, 58], but as researchers attack new program analysis problems, we believe that this perception should be questioned for the following reasons:

- Different client program analyses require different amounts of precision from the pointer analysis [43]. The list of client analyses that have been shown to benefit from flow-sensitive pointer analysis includes several software engineering applications of growing importance, including security analysis [13, 30], deep error checking [34], hardware synthesis [86], and the analysis of multi-threaded programs [72], among others [5, 16, 31].

- The precision of pointer analysis is typically measured in terms of metrics that are averaged over the entire program. In cases such as security analysis and parallelization, these metrics can be misleading—a small amount of imprecision in isolated parts of the program can significantly impact the effectiveness of the client analysis, as demonstrated by Guyer et al. [34]. Thus, two different pointer analyses can have very similar average points-to set sizes but very different impact on the client analysis.

- In a vicious cycle, the lack of an efficient flow-sensitive pointer analysis has inhibited the use of flow-sensitive pointer analyses. The development and widespread use of a scalable flow-sensitive pointer analysis would likely uncover additional client analyses that benefit from the added precision.

- Several techniques [14, 30, 34, 35, 82] can improve the precision of flow-sensitive pointer analysis, but most of these techniques greatly increase the cost of the pointer analysis, making an already non-scalable analysis even more impractical. A significantly more efficient flow-sensitive pointer analysis algorithm would improve the practicality of such techniques, making flow-sensitive pointer analysis even more useful.

Thus, we conclude that there are many reasons to seek a more scalable interprocedural flow-sensitive pointer analysis.

3.1.3 Challenges Facing Flow-Sensitive Pointer Analysis

There are three major performance challenges facing flow-sensitive pointer analysis:

1. **Conservative propagation.** Without pointer information it is in general not possible to determine where variables are defined or used. Therefore, the

analysis must propagate the pointer information generated at each node *k* to all nodes in the CFG reachable from *k* in case those nodes use the information. Typically, however, only a small percentage of the reachable nodes actually require the information, so most of the nodes receive the information needlessly. The effect is to greatly delay the convergence of equations (3.1) and (3.2).

2. **Expensive transfer functions.** Equations (3.1) and (3.2) require a number of set operations with complexity linear in the sizes of the sets involved. These sets tend to be large, with potentially hundreds to thousands of elements. This problem is exacerbated by the analysis' conservative propagation which requires the nodes to needlessly re-evaluate their transfer functions when they receive new pointer information even when that information is irrelevant to the node.

3. **High memory requirements.** Each node in the CFG must maintain two separate points-to graphs, IN for the incoming information and OUT for the outgoing information. For large programs that have hundreds of thousands of nodes, these points-to graphs consume a significant amount of memory. This problem is also exacerbated by the analysis' conservative propagation which requires the IN and OUT graphs to hold pointer information irrelevant to the node in question.

All of the work in improving the scalability of flow-sensitive pointer analysis can be seen as addressing one or more of these challenges. The next section reviews past efforts at meeting these challenges before describing our own solution.

3.2 Related Work

The current state-of-the-art for traditional flow-sensitive pointer analysis using iterative dataflow analysis is described by Hind and Pioli [44, 45], and their analysis is the baseline that we use for evaluating our new techniques. Their analysis employs three major optimizations:

1. **Sparse evaluation graph (SEG) [18, 27, 68].** These graphs are derived from the CFG by eliding nodes that do not manipulate pointer information—and hence are irrelevant to pointer analysis—while maintaining the control-flow relations among the remaining nodes. There are a number of techniques for constructing SEGs, which vary in the complexity of the algorithm and the size of the resulting graph. The use of SEGs addresses challenges (1) and (3) by significantly reducing the input to the analysis.

2. **Priority-based worklist.** Nodes awaiting processing are placed on a worklist prioritized by the topological order of the CFG, such that nodes higher in the CFG are processed before nodes lower in the CFG. This optimization aims to amass at each node as much new incoming pointer information as possible before processing the node, thereby addressing challenge (2) by reducing the number of times the node must be processed.

3. **Filtered forward-binding.** When passing pointer information to the target of a function call, it is unnecessary to pass everything. The only pointer information that the callee can access is that which is accessible from a global or from one of the function parameters. Challenges (1) and (3) can thus be addressed by filtering out the remaining information to add. Less information is propagated unnecessarily, which leads to smaller points-to graphs.

Their evaluation shows that these optimizations speed up the analysis by an average of over 25×. The largest benchmarks analyzed are up to 30,000 lines of code (LOC).

To improve scalability, several non-traditional approaches to flow-sensitive pointer analysis have been proposed. These approaches take inspiration from a number of non-pointer-related program analyses which have addressed similar challenges using a *sparse analysis*, including the use of static single assignment (SSA) form. Pointer analysis cannot directly make use of SSA because pointer information is required to compute SSA form. Cytron et al. [22] propose a scheme for incrementally computing pointer information while converting to SSA form; by incorporating the minimum amount of pointer information necessary, this scheme reduces the size of the resulting SSA form. However, this technique does not speed up the computation of the pointer information itself. We now describe two approaches that use SSA for the actual computation of pointer information.

Hasti and Horwitz [40] propose a scheme composed of two passes: a flow-insensitive pointer analysis that gathers pointer information and a conversion pass that uses the pointer information to transform the program into SSA form. The result of the second pass is iteratively fed back into the first pass until convergence is reached. Hasti and Horwitz leave open the question of whether the resulting pointer information is equivalent to a flow-sensitive analysis; we believe that the resulting information is less precise than a full flow-sensitive pointer analysis. No experimental evaluation of this technique has been published.

Chase et al. [14] propose a technique that dynamically transforms the program to SSA form during the course of the flow-sensitive pointer analysis. There is no experimental evaluation of this proposed technique; however, a similar idea is described and experimentally evaluated by Tok et al. [80]. The technique can ana-

lyze programs that are twice as large as those that use iterative dataflow, enabling the analysis of 70,000 LOC in approximately half-an-hour. Unfortunately, the cost of dynamically computing SSA form limits the scalability of the analysis.

We cannot use a common infrastructure to compare Tok et al.'s technique with ours, because their technique targets programs that begin in non-SSA form, whereas we use the LLVM infrastructure [52], which automatically transforms a program into partial SSA form as described in Section 3.2.1. While the comparison is imperfect due to infrastructure differences, our fastest analysis is 1,286× faster and uses 11.5× less memory on `sendmail`, the only benchmark common to both studies.

A different approach that primarily targets challenges (2) and (3) is symbolic analysis using Binary Decision Diagrams (BDDs), which has been used with great success in model checking [4]. A number of papers have shown that symbolic analysis can greatly improve the performance of flow-insensitive pointer analysis [7, 81, 85, 87]. In addition, Zhu [86] uses BDDs to compute a flow- and context-sensitive pointer analysis for C programs. The analysis is fully symbolic (everything from the CFG to the pointer information is represented using BDDs) but not fully flow-sensitive—the analysis cannot perform indirect strong updates, so the KILL sets are more conservative (i.e., smaller) than a fully flow-sensitive analysis. Symbolic analysis is discussed in more detail in Section 3.3.3. Zhu does not show results for a flow-sensitive, context-insensitive analysis, so we cannot directly compare his techniques with ours.

There have been several other approaches to optimizing flow-sensitive pointer analysis that improve scalability by pruning the input given to the analysis. Rather than improve the scalability of the pointer analysis itself, these techniques reduce the size of its input. Client-driven pointer analysis analyzes the needs of a particu-

lar client and applies flow-sensitive pointer analysis only to portions of the program that require that level of precision [34]. Fink et al. use a similar technique specifically for typestate analysis by successively applying more precise pointer analyses to a program, pruning away portions of the program as each stage of precision has been successfully verified [30]. Kahlon bootstraps the flow-sensitive pointer analysis by using a flow-insensitive pointer analysis to partition the program into sections that can be analyzed independently [47]. These approaches can be combined with our new flow-sensitive pointer analysis to achieve even greater scalability.

3.2.1 SSA

Static single assignment (SSA) form is a common intermediate representation that requires all variables to be defined exactly once in the text of the program. Variables defined multiple times in the original representation are split into separate instances, one for each definition. When separate instances of the same variable are live at a join point in the control-flow graph, they are combined using a ϕ function, which takes the old instances as arguments and assigns the result to a new instance. SSA form is ideal for performing sparse analyses because it makes def-use information explicit in the program representation and allows data-flow information to flow directly from variable definitions to their corresponding uses [69].

There are many known algorithms for converting a program into SSA form [3, 8, 23, 24]. However, the problem becomes more difficult when we consider indirect definitions and uses through pointers. These definitions and uses can only be discovered using pointer analysis. Because of the conservative nature of the pointer analysis results, each indirect definition and use is actually a *possible* definition or use. Following Chow et al. [19], we use χ and μ functions to represent these possible definitions and uses. Assume, without loss of generality, that each indirect

definition corresponds to a STORE instruction and each indirect use corresponds to a LOAD instruction. Each STORE in the original program representation (i.e., prior to the transformation into SSA form) is annotated with a function $v = \chi(v)$ for each variable v that may be defined by the STORE; similarly, each LOAD in the original representation is annotated with a function $\mu(v)$ for each variable v that may be accessed by the LOAD. When converting to SSA form, each χ function is treated as both a definition and use of the given variable, and each μ function is treated as a use of the given variable. The χ function represents the fact that a variable may not be defined at the associated STORE and therefore copies the old value of the variable into the new instance. The way to interpret a STORE with an associated χ function for variable v is that the STORE may define v (in which case its value is the right-hand side of the STORE) or it may not (in which case its value is unchanged).

To avoid these problems, modern compilers such as GCC [62] and LLVM [52] use a variant of SSA, which we call *partial* SSA form. The key idea is to divide variables into two classes. One class contains variables that are never referenced by pointers (*top-level variables*), so their definitions and uses can be trivially determined by inspection without pointer information, and these variables can be converted to SSA using any algorithm for constructing SSA form. The other class contains those variables that *can* be referenced by pointers (*address-taken variables*), and these variables are not placed in SSA form because of the above-mentioned complications.

3.2.1.1 LLVM

Our semi-sparse analysis is implemented in the LLVM infrastructure, so the rest of this section describes LLVM's internal representation (IR) and its particular instantiation of partial SSA form. While the details and terminology are specific to

LLVM, the ideas can be translated to other forms of partial SSA.

LLVM's IR recognizes two classes of variables: (1) *top-level* variables are those that cannot be referenced indirectly via a pointer, i.e., those whose address is never exposed via the address-of operator or returned via a dynamic memory allocation; (2) *address-taken* variables are those that have had their address exposed and therefore can be indirectly referenced via a pointer. Top-level variables are kept in a (conceptually) infinite set of virtual registers which are maintained in SSA form. Address-taken variables are kept in memory, and they are not in SSA form. Address-taken variables are accessed via LOAD and STORE instructions, which take top-level pointer variables as arguments. These address-taken variables are never referenced syntactically in the IR; they instead are only referenced indirectly using these LOAD and STORE instructions. LLVM instructions use a 3-address format, so there is at most one level of pointer dereference for each instruction.

Figure 3.1 provides an example of a C code fragment and its corresponding partial SSA form. Variables `w`, `x`, `y`, and `z` are top-level variables and have been converted to SSA form; variables `a`, `b`, `c`, and `d` are address-taken variables, so they are stored in memory and accessed solely via LOAD and STORE instructions. Because the address-taken variables are not in SSA form, they can each be defined multiple times, as with variables `c` and `d` in the example.

Because address-taken variables cannot be directly named, LLVM maintains the invariant that each address-taken variable has at least one virtual register that refers only to that variable. To illustrate this point, Figure 3.2 shows how a temporary variable, `t`, is introduced in the LLVM IR to take the place of the variable `b`, which in the original C code is referenced by a pointer.

LLVM also treats global variables specially. Def-use chains for global variables can span multiple functions; however, in the presence of indirect function

```
int a, b, *c, *d;

int*  w = &a;          w1 = ALLOCa
int*  x = &b;          x1 = ALLOCb
int** y = &c;          y1 = ALLOCc
int** z =  y;          z1 = y1
      c =  0;          STORE 0 y1
     *y =  w;          STORE w1 y1
     *z =  x;          STORE x1 z1
      y = &d;          y2 = ALLOCd
      z =  y;          z2 = y2
     *y =  w;          STORE w1 y2
     *z =  x;          STORE x1 z2
```

Figure 3.1: Example partial SSA code. On the left is the original C code, on the right is the transformed code in partial SSA form.

```
int **a, *b, c;
      a = &b;          a = ALLOCb
      b = &c;          t = ALLOCc
      c = 0;           STORE t a
                       STORE 0 t
```

Figure 3.2: Example partial SSA code. On the left is the original C code, on the right is the transformed code in partial SSA form.

calls it is not possible to construct precise def-use chains across function boundaries without pointer information. To address this issue, LLVM adds an extra level of indirection to each global variable: `T glob` becomes `const T* glob`, where `T` is the type of the global declared in the original program. The const pointers are initialized to point to an address-taken variable that represents the original global variable. This modification means that pointer information for global variables is propagated along the SEG rather than relying on cross-function def-use chains.

Note: The rest of this chapter will assume the use of the LLVM IR, which means that any named variable is a top-level variable and not an address-taken variable.

3.3 Semi-Sparse Analysis

For flow-sensitive pointer analysis, partial SSA form has the following important implications that have not been previously identified or explored.

1. The analysis can use a single global points-to graph to hold the pointer information for all top-level variables. Since the variables are in SSA form, they will necessarily have the same pointer information over the entire program. The presence of this global points-to graph means the analysis can avoid storing and propagating the pointer information for top-level variables among CFG nodes.

2. Def-use information for top-level variables is immediately available, as in a sparse analysis. When pointer information for a top-level variable changes, the affected program statements can be directly determined, which can dramatically speed up the convergence of the analysis and reduce the number of transfer functions that must be evaluated.

3. Local points-to graphs, i.e., separate IN and OUT graphs for each CFG node, are still needed for LOAD and STORE statements, but these graphs only hold pointer information for address-taken variables. The exclusion of top-level variables can significantly reduce the sizes of these local points-to graphs.

Semi-sparse analysis takes advantage of partial SSA form to greatly increase the efficiency of the flow-sensitive pointer analysis. In order to do so, we introduce a construct called the *Dataflow Graph*. We first describe the characteristics of the dataflow graph and how it is constructed, and we then describe the semi-sparse analysis itself, followed by the new optimizations enabled by partial SSA.

3.3.1 The Dataflow Graph

The dataflow graph (DFG) is a combination of a sparse evaluation graph (SEG) and def-use chains. This combination is required by the nature of partial SSA form, which provides def-use information for the top-level variables but not for the address-taken variables.

Without access to def-use information, an iterative dataflow analysis propagates information along the control-flow graph. As described in Section 3.2, the SEG is simply an optimized version of the control-flow graph that elides nodes that neither define nor use pointer information. Since address-taken variables do not have def-use information available, program statements that define or use address-taken variables must be connected via a path in the SEG so that variable definitions will correctly reach their corresponding uses. Since top-level variables have def-use information immediately available, program statements that define or use top-level variables can be connected via these def-use chains.

To construct the DFG there are 6 types of relevant program statements,

Inst Type	Example	Def-Use Info
ALLOC	`x = ALLOC`$_i$	DEF$_{top}$
COPY	`x = y z`	DEF$_{top}$, USE$_{top}$
LOAD	`x = *y`	DEF$_{top}$, USE$_{top}$, USE$_{adr}$
STORE	`*x = y`	USE$_{top}$, DEF$_{adr}$, USE$_{adr}$
CALL	`x = foo(y)`	DEF$_{top}$, USE$_{top}$, DEF$_{adr}$, USE$_{adr}$
RET	`return x`	USE$_{top}$, USE$_{adr}$

Table 3.1: Types of instructions relevant to pointer analysis. Instructions such as `x = &y` are converted into ALLOC instructions, much like C's alloca. *Def-Use Info* describes whether the instruction can define or use top-level variables (DEF$_{top}$ and USE$_{top}$, respectively) and whether it can define or use address-taken variables (DEF$_{adr}$ and USE$_{adr}$, respectively). Recall that all named variables are, by construction, top-level.

shown in Table 3.1. For each statement, the table lists whether it defines and/or uses top-level variables (DEF$_{top}$ and USE$_{top}$, respectively) and whether the statement defines and/or uses address-taken variables (DEF$_{adr}$ and USE$_{adr}$, respectively). STORE instructions are labeled USE$_{adr}$ because weak updates require the updated variable's previous points-to set. CALL instructions are labeled DEF$_{adr}$ because they can modify address-taken variables via the callee function. CALL and RET instructions are labeled USE$_{adr}$ because they need to pass the address-taken pointer information to/from the callee function. COPY instructions can have multiple variables on the right-hand side, which allows them to accommodate SSA ϕ functions.

The DFG is constructed in two stages. In the first stage, a standard algorithm for creating an SEG (such as Ramalingam's linear-time algorithm [68]) is used. Only program statements labeled DEF$_{adr}$ or USE$_{adr}$ are considered relevant; all others are elided. Then a linear pass through the partial SSA representation is used to connect program statements that define top-level variables with those that use those variables. Figure 3.3 shows the DFG corresponding to the partial SSA

code in Figure 3.1.

Theorem 1 (Correctness of the DFG). There exists a path in the DFG from all variable definitions to their corresponding uses.

Proof. We proceed by cases based on the type of variable:

Top-level: Def-use information for top-level variables is exposed by the partial SSA form; the DFG directly connects top-level variable definitions to their uses, so the theorem is trivially true.

Address-taken: All uses of a variable's definition must be reachable from the statement that created the definition in the original control-flow graph. The SEG preserves control-flow information for all statements that either define or use address-taken variables. Therefore any use of an address-taken variable's definition must be reachable from the statement that created the definition in the SEG.

□

3.3.2 The Analysis

The pointer analysis itself is similar to that described by Hind and Pioli [44, 45]. The analysis uses the following data structures:

- Each function F has its own program statement worklist $StmtWorklist_F$. The worklist is initialized to contain all statements in the function that define a variable (i.e., are labeled DEF_{adr} or DEF_{top}).

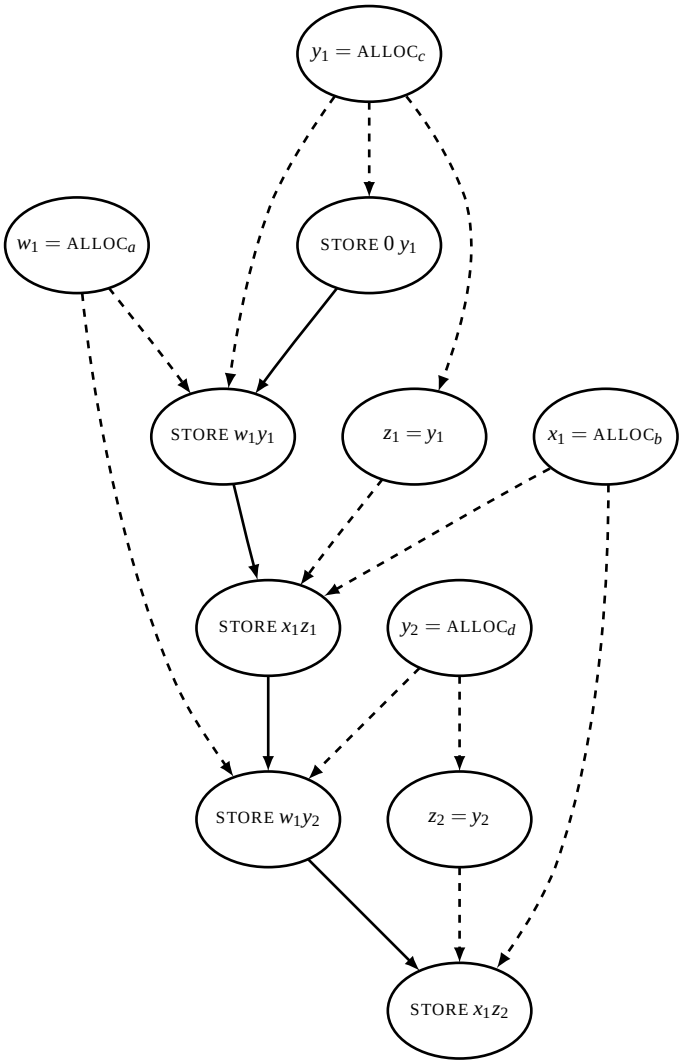


Figure 3.3: Example DFG corresponding to the code in Figure 3.1. Dashed edges are def-use chains; solid edges are for the SEG.

- Each program statement k that uses or defines address-taken variables (i.e., is labeled USE_{adr} or DEF_{adr}) has two points-to graphs, IN_k and OUT_k, which hold the incoming and outgoing pointer information for address-taken variables. Let $\mathcal{P}_k(\mathsf{v})$ be the points-to set of address-taken variable v in IN_k.

- A global points-to graph PG_{top} holds the pointer information for all top-level variables. Let $\mathcal{P}_{top}(\mathsf{v})$ be the points-to set of top-level variable v in PG_{top}.

- A worklist *FunctionWorklist* holds functions waiting to be processed. The worklist is initialized to contain all functions in the program.

The main body of the analysis is listed in Algorithm 4. The outer loop selects a function from the function worklist, and the inner loop iteratively selects a program statement from that function's statement worklist and processes it, continuing until the statement worklist is empty. Then the analysis selects a new function from the function worklist, continuing until the function worklist is also empty. Each type of program statement is processed as shown in Algorithms 8–13. These algorithms use the helper functions listed in Algorithms 5–7. The $\hookleftarrow$ operator represents set update; $\xrightarrow{du}$ and $\xrightarrow{SEG}$ represent a def-use edge or SEG edge in the DFG, respectively.

3.3.2.1 Optimizations

Partial SSA form allows us to introduce two additional optimization opportunities: *top-level pointer equivalence* and *local points-to graph equivalence*.

Top-level Pointer Equivalence Top-level pointer equivalence reduces the number of top-level variables in the DFG, which reduces the amount of pointer information that must be maintained by the global top-level points-to graph. In addition,

Algorithm 4 Main body of the semi-sparse analysis algorithm.

Require: $DFG = \langle N, E \rangle$
  while *FunctionWorklist* is not empty **do**
    F =SELECT(*FunctionWorklist*)
    while $StmtWorklist_F$ is not empty **do**
      k =SELECT($StmtWorklist_F$)
      switch *typeof*(k)**:**
        case ALLOC: processAlloc(F,k)
        case COPY: processCopy(F,k)
        case LOAD: processLoad(F,k)
        case STORE: processStore(F,k)
        case CALL: processCall(F,k)
        case RET: processRet(F,k)

Algorithm 5 propagateTopLevel(F,k)
  if PG_{top} changed **then**
    $StmtWorklist_F \hookleftarrow \{\, n \mid k \xrightarrow{du} n \in E \}$

Algorithm 6 propagateAddrTaken(F,k)
  for all $\{n \in N \mid k \xrightarrow{SEG} n \in E\}$ **do**
    $\text{IN}_n \hookleftarrow \text{OUT}_k$
    if IN_n changed **then**
      $StmtWorklist_F \hookleftarrow \{n\}$

Algorithm 7 filter(k)
  return the subset of IN_k reachable from either a call argument or global variable

Algorithm 8 processAlloc(F,k) : [`x = ` ALLOC$_i$]
  $PG_{top} \hookleftarrow \{\texttt{x} \rightarrow \text{ALLOC}_i\}$
  propagateTopLevel(F,k)

Algorithm 9 processCopy(F,k) : [`x = y z ...`]
  for all $v \in$ right-hand side **do**
    $PG_{top} \hookleftarrow \{\texttt{x} \rightarrow \mathcal{P}_{top}(v)\}$
  propagateTopLevel(F,k)

Algorithm 10 processLoad(F, k) : [`x = *y`]

$PG_{top} \hookleftarrow \{\mathtt{x} \rightarrow \mathcal{P}_k(\mathcal{P}_{top}(\mathtt{y}))\}$
$\text{OUT}_k \hookleftarrow \text{IN}_k$
propagateTopLevel(F, k)
propagateAddrTaken(F, k)

Algorithm 11 processStore(F, k) : [`*x = y`]

if $\mathcal{P}_{top}(\mathtt{x})$ represents a single memory location **then**
  // *strong update*
  $\text{OUT}_k \hookleftarrow (\text{IN}_k \setminus \mathcal{P}_{top}(\mathtt{x})) \cup \{\mathcal{P}_{top}(\mathtt{x}) \rightarrow \mathcal{P}_{top}(\mathtt{y})\}$
else // *weak update*
  $\text{OUT}_k \hookleftarrow \text{IN}_k \cup \{\mathcal{P}_{top}(\mathtt{x}) \rightarrow \mathcal{P}_{top}(\mathtt{y})\}$
propagateAddrTaken(F, k)

Algorithm 12 processCall(F, k) : [`x = foo(y)`]

if `foo` is a function pointer **then**
  targets := $\mathcal{P}_{top}(\mathtt{foo})$
else
  targets := $\{\mathtt{foo}\}$
filt := filter(k)
for all $C \in$ *targets* **do**
  for all call arguments `a` and corresponding parameters `p` **do**
    $PG_{top} \hookleftarrow \{\mathtt{p} \rightarrow \mathcal{P}_{top}(\mathtt{a})\}$
    propagateTopLevel($C, \mathtt{p}$)
  Let n be the SEG start node for function C
  $\text{IN}_n \hookleftarrow$ *filt*
  if IN_n changed **then**
    StmtWorklist$_C \hookleftarrow \{n\}$
  if *StmtWorklist*$_C$ changed **then**
    FunctionWorklist $\hookleftarrow \{C\}$
$\text{OUT}_k \hookleftarrow \text{IN}_k \setminus$ *filt*
propagateAddrTaken(F, k)

Algorithm 13 processRet(F, k) : [`return x`]

```
callsites := the set of CALL statements targeting F
for all n ∈ callsites do
    Let F_n be the function containing n
    OUT_n ↩ OUT_k
    propagateAddrTaken(F_n, n)
    if n is of the form r = F(...) then
        PG_top ↩ {r → P_top(x)}
        propagateTopLevel(F_n, n)
    if StmtWorklist_{F_n} changed then
        FunctionWorklist ↩ {F_n}
```

it eliminates nodes from the DFG, which reduces the number of transfer functions that must be processed, speeding up convergence. The basic idea is to identify sets of variables that have identical points-to sets and to replace each set by a single set representative.

Pointer equivalent variables are those that have identical points-to sets. More formally, let $\rightarrow$ be the points-to relation and $\bowtie$ be the pointer equivalence relation; then $\forall x, y, z \in \mathit{Variables} : x \bowtie y$ iff $x \rightarrow z \Leftrightarrow y \rightarrow z$. Program variables can be partitioned into disjoint sets based on the pointer equivalence relation; an arbitrary member of each set is then selected as the set representative. By replacing all variables in a program with their respective set representatives and then eliding trivial assignments (e.g., `x = x`), we can reduce the number of variables and the size of the program that are given as input to the pointer analysis. This idea has been previously explored for flow-insensitive pointer analysis [38, 70].

Partial SSA form provides an opportunity to apply this optimization to flow-sensitive pointer analysis as well. To do so, we must be able to identify pointer-equivalent variables prior to the pointer analysis itself. Theorem 2 shows how we can identify top-level pointer-equivalent variables under certain circumstances.

Theorem 2 (Top-level pointer equivalence). A COPY statement of the form $[\mathsf{x} = \mathsf{y}] \Rightarrow \mathsf{x} \bowtie \mathsf{y}$.

Proof. Top-level variables are in SSA form, which means that they are each defined exactly once. Therefore, the value of each top-level variable does not change once it is defined.

Since x and y are top-level variables, their values never change. The COPY statement assigns x the value of y, so $\mathsf{x} \bowtie \mathsf{y}$. □

Theorem 2 says that variables involved in a COPY statement with a single variable on the right-hand side are pointer equivalent, so they can be replaced with a single representative variable. The COPY statement (called a *single-use* COPY) is then redundant and can be discarded from the DFG. When statements are discarded, any edges to those statements must be updated to point to the successors of the discarded statement. If node n is discarded from $DFG = \langle N, E \rangle$ then the result is a new $DFG = \langle N', E' \rangle$ where:

- $N' = N \setminus \{n\}$
- $E' = E \setminus \{k \to n\} \cup \{k \to p \mid \{k \to n, n \to p\} \subseteq E\}$

In Figure 3.3, $\mathsf{y}_1 \bowtie \mathsf{z}_1$ and $\mathsf{y}_2 \bowtie \mathsf{z}_2$. We can replace all occurrences of z_1 with y_1, replace all occurrences of z_2 with y_2, and eliminate the nodes for $[\mathsf{z}_1 = \mathsf{y}_1]$ and $[\mathsf{z}_2 = \mathsf{y}_2]$. The def-use edge from $[\mathsf{y}_1 = \text{ALLOC}_c]$ to $[\mathsf{z}_1 = \mathsf{y}_1]$ is removed, and a new def-use edge is added from $[\mathsf{y}_1 = \text{ALLOC}_c]$ to $[\text{STORE } \mathsf{x}_1\ \mathsf{y}_1]$. Similarly, the def-use edge from $[\mathsf{y}_2 = \text{ALLOC}_d]$ to $[\mathsf{z}_2 = \mathsf{y}_2]$ is removed, and a new def-use edge is added from $[\mathsf{y}_2 = \text{ALLOC}_d]$ to $[\text{STORE } \mathsf{x}_1\ \mathsf{y}_2]$. Figure 3.4 shows the optimized version of Figure 3.3.

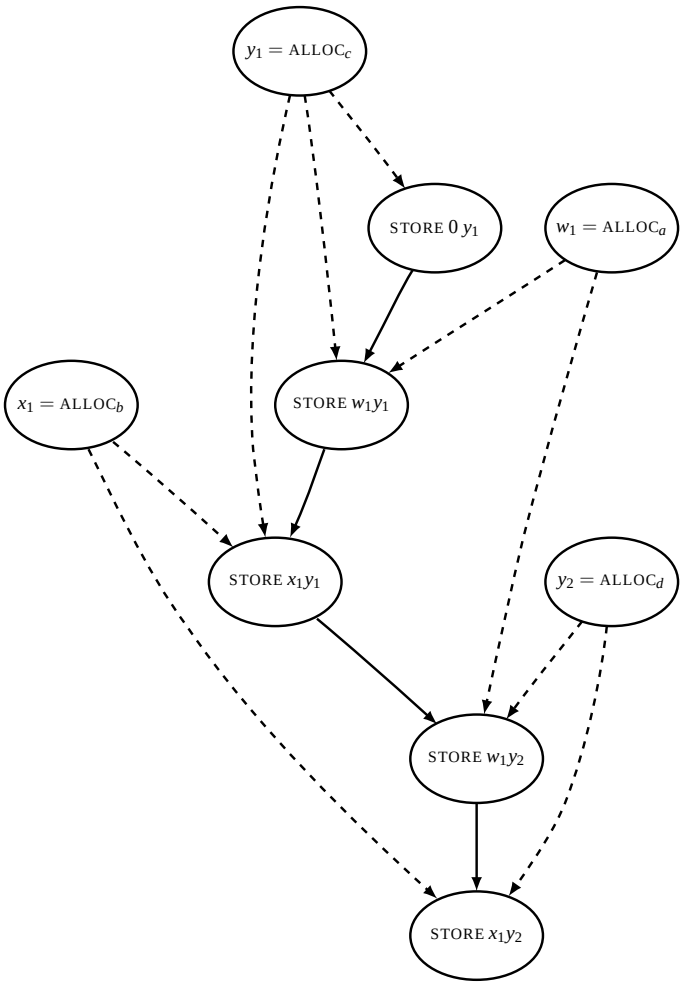


Figure 3.4: Figure 3.3 optimized using top-level pointer equivalence.

Theorem 3 (Correctness of the Transformation). The top-level pointer equivalence transformation preserves SSA form for top-level variables.

Proof. There are two characteristics of SSA form that the transformation must preserve:

Every variable is defined exactly once. Let V be a set of pointer-equivalent variables found by the transformation and let S be the set of statements that define these variables. S contains exactly one statement that is not a single-use COPY. S must contain at least one such statement because otherwise S forms a cycle in the def-use graph such that a variable is used before it is defined, which would violate SSA form. S cannot contain more than one such statement because only single-use COPYs are considered when finding equivalent variables. After the equivalent variables are replaced by their set representative, all of the single-use COPYs in S are deleted, leaving exactly one statement that defines the representative variable.

Every definition dominates all of its uses. Every single-use COPY in S is dominated by a statement in S—if a statement $\mathsf{x} = \mathsf{y} \in S$, then $\mathsf{x}, \mathsf{y} \in V$ and by definition S must also contain the statement defining y. There is exactly one statement in S that is not a single-use COPY; therefore that statement must dominate all other statements in S. When the single-use COPYs are deleted, all of the edges pointing to those statements are updated as described above—therefore the remaining statement in S must dominate all statements in the program that used a variable in V. □

Local Points-to Graph Equivalence Local points-to graph equivalence allows nodes in the DFG that are guaranteed to have identical points-to graphs to share a single graph rather than maintain separate copies. This sharing can significantly re-

duce the memory consumption of the pointer analysis, as well as reduce the number of times pointer information must be propagated among nodes.

To identify nodes with identical points-to graphs, we define the notion of *non-preserving* nodes. The points-to graphs that are local to nodes in the DFG (i.e., IN_k and OUT_k) only contain pointer information for address-taken variables. By the nature of partial SSA form, only STORE instructions and CALL instructions (which reflect the changes caused by STORE instructions in the callee function) can modify the address-taken pointer information; we call these nodes *non-preserving*. Other instructions may use this information (e.g., LOAD and RET instructions), but they propagate the pointer information through the DFG unchanged; we call these nodes *preserving*. We say that non-preserving node p *reaches* node q ($p \rightsquigarrow q$) if there is a path in the DFG from p to q, using only SEG edges, that does not contain a non-preserving node. There may be a number of nodes in the DFG that are all reachable from the same set of non-preserving nodes; Theorem 4 says that these nodes are guaranteed to have identical points-to graphs.

Theorem 4 (Local points-to graph equivalence). Let $N_{np} \subseteq N$ be the set of non-preserving DFG nodes. $\forall p \in N_{np}$ and $q, r \in N : (p \rightsquigarrow r \Leftrightarrow p \rightsquigarrow q) \Rightarrow q$ and r have identical points-to graphs.

Proof. Assume $\exists q, r \in N.(\forall p \in N_{np} : p \rightsquigarrow q \Leftrightarrow p \rightsquigarrow r)$, and that q and r do not have identical points-to graphs. Then one of the nodes (assume it is q) must have received pointer information that the other did not. However, by construction of the partial SSA form, non-preserving nodes are the only places that can generate new pointer information for address-taken variables (the only kind of variable present in the local points-to graphs). Therefore $\exists p \in N_{np}.(p \rightsquigarrow q \wedge \neg(p \rightsquigarrow r))$. But this violates our initial assumption that both p and q are reachable from the same set of non-preserving nodes. Therefore, p and q must have identical points-to graphs. □

A simple algorithm (see Algorithm 14) can detect nodes that can share their points-to graphs. For each STORE and CALL node in the DFG, the algorithm labels all nodes that are reachable via a sequence of SEG edges without going through another STORE or CALL node with a label unique to the originating node. Since nodes may be reached by more than one STORE or CALL node, each node will end up with a set of labels. This process takes $O(n^3)$ time, where n is the number of nodes in the SEG portion of the DFG. These labels represent the propagation of the unknown pointer information computed by the originating node. All nodes with an identical set of labels are guaranteed to have identical local points-to graphs and can therefore share a single graph among them.

Algorithm 14 Detecting nodes with equivalent points-to graphs.

Require: $DFG = \langle N, E \rangle$
Require: $\forall n \in N : id_n$ is a unique identifier
Require: $Worklist = N$
  while *Worklist* is not empty **do**
    $n =$SELECT$(Worklist)$
    for all $\{k \in N \mid k \xrightarrow{SEG} n \in E\}$ **do**
      if $typeof(k) \in \{$STORE, CALL$\}$ **then**
        $label_n \hookleftarrow \{id_k\}$
      else
        $label_n \hookleftarrow label_k$
      if $label_n$ changed **then**
        for all $\{p \in N \mid n \xrightarrow{SEG} p \in E\}$ **do**
          $Worklist \hookleftarrow \{p\}$

By potentially sacrificing a small amount of precision, we can greatly increase the effectiveness of this optimization. CALL nodes turn out to be a large percentage of the total number of nodes in the DFG. By assuming that callees do not modify address-taken pointer information accessible by their callers, thereby allowing Algorithm 14 to treat CALL nodes exactly the same as all other non-STORE

nodes, we can significantly increase the amount of sharing between nodes. This assumption is sound—the optimization only causes nodes to share points-to graphs, so if a callee does modify address-taken pointer information, the pointer information is propagated to additional nodes that it otherwise wouldn't have reached. The effect of this assumption on precision and performance is explored in Section 3.3.4.

3.3.3 Symbolic Analysis

This section briefly discusses the pros and cons of using Binary Decision Diagrams (BDDs) for flow-sensitive pointer analysis. BDDs are data structures for compactly representing sets and relations [10]. BDDs have several advantages over other data structures for this purpose: (1) the size of a BDD is only loosely correlated with the number of elements in the set that the BDD represents, meaning that large sets can be stored in very little space, and (2) the complexity of set operations involving BDDs depends only on the sizes of the BDDs involved, not on the number of elements in the sets. *Symbolic analysis* takes advantage of these characteristics to perform analyses that would be prohibitively expensive—both in time and memory—using more conventional data structures. There are a number of examples of symbolic pointer analyses in the literature [7, 81, 85–87]. These analyses are fully symbolic: all relevant information is stored as either a set or relation using BDDs, and the analysis is completely expressed in terms of operations on those BDDs. When applied specifically to flow-sensitive pointer analysis [86], the relevant information is the control-flow graph and the points-to relations; these are stored in BDDs and the transfer functions for the CFG nodes are expressed as BDD operations. Thus, the analysis essentially computes the transfer functions for all nodes in the CFG simultaneously, making the analysis very efficient.

The strength of symbolic analysis lies in its ability to quickly perform op-

erations on entire sets. Its weakness is that it is not well-suited for operating on individual members of a set independently from each other. This weakness directly impacts flow-sensitive pointer analysis. The KILL sets for indirect assignments, such as `*x = y`, cannot be efficiently computed on-the-fly because their contents depend not only on the pointer information computed during the analysis itself but also on the individual characteristics of the points-to set elements at the node in question, e.g., whether a particular element represents a single memory location or multiple memory locations (as would be true for a variable summarizing the heap). Therefore a fully symbolic flow-sensitive pointer analysis must either process each indirect assignment separately, at prohibitive cost, or conservatively set all KILL sets for indirect assignments to the empty set, sacrificing precision.

We propose an alternative to a fully symbolic analysis, which is to encode only a subset of the problem using BDDs. For pointer analysis the most useful subset to encode is the set of points-to relations, which is responsible for the vast majority of both memory consumption and set operations in the analysis. By isolating the pointer information representation into its own source code module, we can easily substitute a BDD-based implementation while leaving the rest of the analysis completely unchanged, including the on-the-fly computation of KILL sets. In our experimental evaluation we study the effects of using BDDs to represent pointer information for both the baseline analysis (based on Hind and Pioli [45]) and our new semi-sparse analysis.

3.3.4 Evaluation

To evaluate our new techniques, we implement three flow-sensitive pointer analysis algorithms: a baseline analysis based on Hind and Pioli [45] (IFS); semi-sparse flow-sensitive analysis (SS); and the semi-sparse analysis augmented with

our two new optimizations, top-level pointer equivalence and local points-to graph equivalence (SSO). All the algorithms are field-sensitive (i.e., they treat each field of a struct as a separate variable). For each algorithm, we evaluate two versions, one that implements pointer information using sparse bitmaps and a second that uses BDDs. We imlement the algorithms in the LLVM compiler infrastructure [52]. The BDDs use the BuDDy BDD library [54]. The algorithms are written in C++ and handle all aspects of the C language except for varargs.

The bitmap versions of IFS, SS, and SSO filter pointer information at call-sites as described by Hind and Pioli (see Section 3.2 and Section 3.3.2). The BDD versions of these algorithms do not use filtering. The goal of filtering is to reduce the amount of pointer information propagated between callers and callees in order to speed up convergence and reduce the sizes of the points-to graphs. As mentioned earlier, with BDDs we don't need to worry about the sizes of the points-to graphs, and in fact for the BDD versions, the overhead involved in filtering the pointer information overwhelms any potential benefit.

The benchmarks for our experiments are described in Table 3.2. Six of the benchmarks are taken from SPECINT 2000[1] (the largest six applications from that suite: parser, twolf, vortex, gap, perlbmk, and gcc) and six from various open-source applications. For the non-SPEC benchmarks: ex is a text processor; send-mail is an email server; vim is a text processor; nethack is a text-based game; gdn is a C language debugger; and ghostscript is a postscript viewer. Function calls to external code are summarized using hand-crafted function stubs. The experiments are run on a 1.83 GHz processor with 2 GB of memory, using the Ubuntu 7.04 Linux distribution.

[1] www.spec.org/cpu2000/ as of 5/2009/

Name	LOC	Statements	Functions	Call Sites
parser	11.4K	33.6K	99	774
ex-050325	34.4K	37.0K	325	2,519
twolf	20.5K	45.0K	107	331
vortex	67.2K	69.2K	271	4,420
sendmail-8.11.6	88.0K	69.3K	273	3,203
gap	71.4K	132.2K	725	6,002
perlbmk	85.5K	184.6K	726	8,597
vim-7.1	323.5K	316.4K	1,935	15,962
nethack-3.4.3	252.6K	356.3K	1,385	23,001
gcc	226.5K	376.2K	1,159	19,964
gdb-6.7.1	474.1K	484.3K	3,801	37,119
ghostscript-8.15	429.0K	494.0K	4,815	18,050

Table 3.2: Benchmarks: lines of code (**LOC**) is obtained by running `wc` on the source. **Statements** reports the number of statements in the LLVM IR. The benchmarks are ordered by number of statements.

3.3.4.1 Performance Results

Tables 3.3 and 3.4 give the analysis time and memory consumption of the various algorithms. These numbers include the time to build the data structures, apply the optimizations, and compute the pointer analysis.

For the bitmap versions of these algorithms, memory is the limiting factor. IFS only scales to 20.5K LOC before running out of memory, SS scales to 67.2K LOC, and SSO scales to 252.6K LOC. For the two benchmarks that IFS manages to complete, SS is 75× faster and uses 26× less memory, while SSO is 183× faster and uses 47× less memory. For the four benchmarks that SS completes, SSO is 2.5× faster and uses 6.8× less memory.

For the BDD versions of these algorithms, memory is not an issue and all three algorithms scale to 323.5K LOC. However, the two largest benchmarks (`gdb`

Name	IFS		SS		SSO	
	time	mem	time	mem	time	mem
197.parser	80.25	888	1.28	53	0.52	15
ex-050325	—	OOM	15.74	198	7.33	39
300.twolf	72.28	415	0.82	32	0.34	12
255.vortex	—	OOM	33.37	1,275	11.70	81
sendmail-8.11.6	—	OOM	—	OOM	86.38	258
254.gap	—	OOM	—	OOM	191.72	518
253.perlbmk	—	OOM	—	OOM	—	OOM
vim-7.1	—	OOM	—	OOM	—	OOM
nethack-3.4.3	—	OOM	—	OOM	4,762.07	1,648
176.gcc	—	OOM	—	OOM	—	OOM
gdb-6.7.1	—	OOM	—	OOM	—	OOM
ghostscript-8.15	—	OOM	—	OOM	—	OOM

Table 3.3: Performance: time (in seconds) and memory consumption (in megabytes) of the various analyses using bitmaps. OOM means the benchmark ran out of memory.

and `ghostscript`) do not complete within our arbitrary time limit of eight hours. For the ten benchmarks that they do complete, SS is 44.8× faster than IFS and uses 1.4× less memory, while SSO is 114× faster and uses 1.4× less memory. Comparing the fastest algorithm in our study (SSO using BDDs) with our baseline algorithm (IFS using bitmaps) using the two benchmarks that IFS manages to complete, we have sped up flow-sensitive analysis 197× while using 4.6× less memory.

Figures 3.5 and 3.6 describe various analysis statistics to explain the relative performance of these algorithms. Figure 3.5 gives the percentage of points-to graphs that SS and SSO have compared to IFS (i.e., the number of points-to graphs maintained at each node summed over all the nodes). Figure 3.6 gives the percentage of instructions that are processed by SS and SSO compared to IFS (i.e., the total number of nodes popped off of the statement worklists in Algorithm 4).

Name	IFS		SS		SSO	
	time	mem	time	mem	time	mem
197.parser	7.24	142	0.64	142	0.48	142
ex-050325	7.95	142	0.66	143	0.46	142
300.twolf	6.41	143	0.46	144	0.32	143
255.vortex	14.39	150	0.97	151	0.78	150
sendmail-8.11.6	38.51	150	2.16	154	1.40	152
254.gap	68.66	167	2.50	168	2.34	166
253.perlbmk	1,477.05	280	50.22	182	21.25	177
vim-7.1	4,759.37	535	573.28	300	112.16	263
nethack-3.4.3	3,435.48	423	13.68	225	5.37	220
176.gcc	2,445.27	595	39.71	234	9.37	226
gdb-6.7.1	OOT	—	OOT	—	OOT	—
ghostscript-8.15	OOT	—	OOT	—	OOT	—

Table 3.4: Performance: time (in seconds) and memory consumption (in megabytes) of the various analyses using BDDs. OOT means the analysis ran out of time (exceeded an eight hour time limit).

For IFS the pointer-related instructions have been grouped into basic blocks to reduce the number of points-to graphs that need to be maintained. This grouping is not possible for SS and SSO because they have def-use chains between individual instructions. However, averaged over all the benchmarks, SS still has 24.6% fewer points-to graphs than IFS because only nodes in the SEG portion of the dataflow graph require points-to graphs. Also recall that the points-to graphs for SS and SSO only have to hold pointer information for address-taken variables, so they are much smaller than the points-to graphs for IFS. SSO reduces the number of points-to graphs by another 66.6% over SS using local points-to graph equivalence.

The use of top-level def-use chains for semi-sparse analysis pays off: averaged over all the benchmarks, SS processes 62.9% fewer instructions than IFS. SSO further reduces the number of instructions processed by 13.7% over SS.

3.3.4.2 Performance Discussion

Semi-sparse analysis delivers on its promise. Based on the number of instructions processed and the reported efficiency, semi-sparse analysis significantly speeds up convergence. When using bitmaps, the global top-level points-to graph significantly reduces memory consumption as well, especially when coupled with the top-level pointer equivalence and local points-to graph equivalence optimizations. However, there are some results which may be a bit surprising; we highlight these results and explain them in this section.

First, note the memory requirements for the BDD analyses as compared to the sparse bitmap analyses. We see for the smaller benchmarks that the BDDs actually require more memory than the bitmaps, even though the premise behind BDDs is that they are more memory efficient. This discrepancy arises because of the implementation of the BuDDy library—an initial pool of memory is allocated

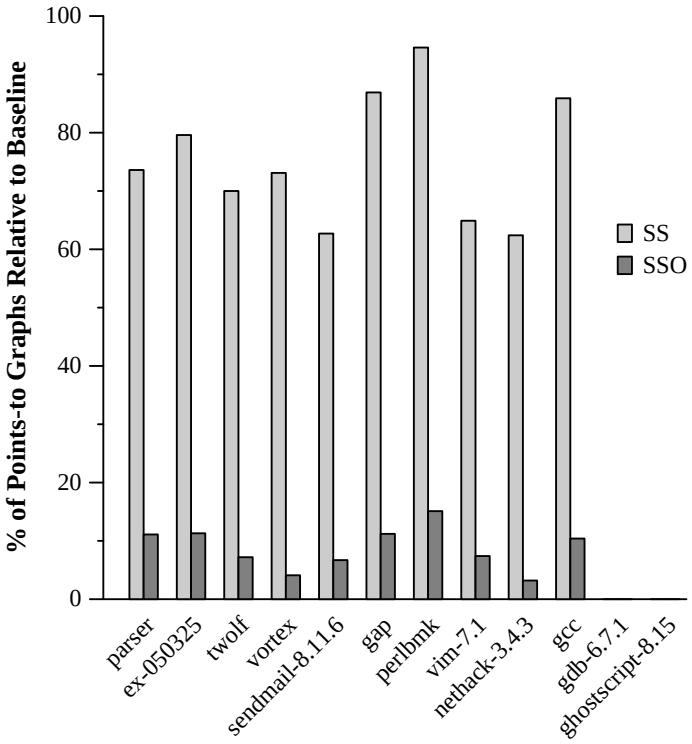


Figure 3.5: Number of points-to graphs maintained by SS and SSO compared to IFS. Lower is better (fewer points-to graphs).

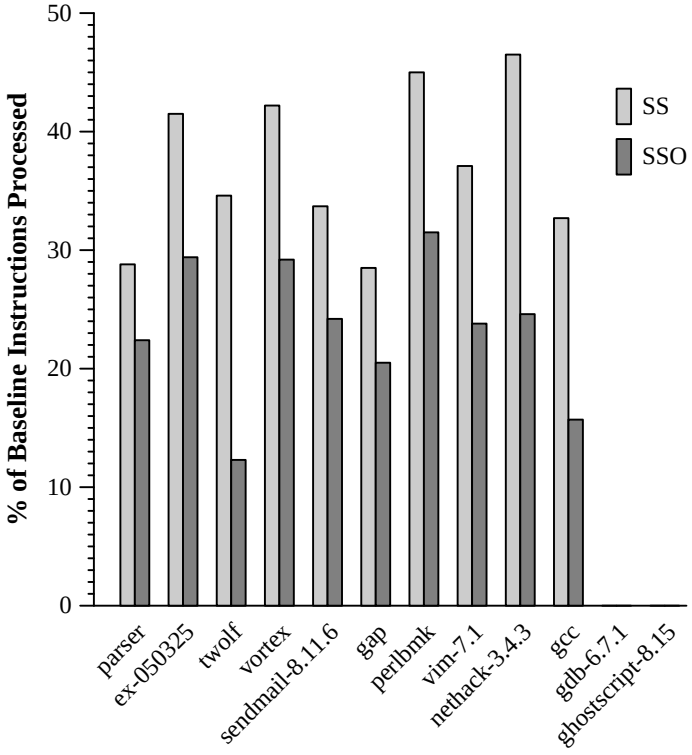


Figure 3.6: Number of instructions processed by SS and SSO compared to IFS. Lower is better (fewer instructions processed).

before the analysis begins, then expanded as necessary. On the larger benchmarks, we see that the memory requirements for the BDD analyses rise much more slowly than that for the bitmaps, bearing out our initial premise.

Second, the bitmap version of SSO completes for nethack-3.4.3, but runs out of memory for two benchmarks with fewer statements (253.perlbmk and vim-7.1). This showcases the difficulty of predicting analysis performance based solely on the input size—the actual performance of the analysis also depends on factors that are impossible to predict before the analysis is complete, such as the points-to set sizes of the variables and how widely the pointer information is dispersed via indirect calls.

Third, the time required for the SS and SSO BDD analyses to analyze 253.perlbmk, vim-7.1, gdb-6.7.1, and ghostscript-8.15 seem disproportionately long considering the analysis times for the other benchmarks. There is one minor and one major reason for this anomaly. The minor reason is specific to 253.perlbmk—the field-sensitive solution has an average points-to set size over twice that of the field-insensitive solution. This result seems counter-intuitive, since field-sensitivity should add precision and hence reduce points-to set size. However, to account for the individual fields of the structs, field-sensitive analysis increases the number of address-taken variables, in some cases (such as 253.perlbmk) making the points-to set sizes larger than for a field-insensitive analysis, even though the analysis results are, in fact, more precise. With the exception of 253.perlbmk, all the other benchmarks do have smaller points-to set sizes for the field-sensitive analysis.

For the remaining three benchmarks with disproportionately large analysis times (vim-7.1, gdb-6.7.1, and ghostscript-8.15), the major reason for the anomaly is the BDDs themselves. To confirm this finding, we measure the average processing time per node for each of the benchmarks and find that these three benchmarks

have a much higher time per node than the others. The main cost of processing a node is the manipulation of pointer information, which points out a weakness of BDDs—their performance is directly related to how well they compact the information that they are storing, and it is impossible to determine *a priori* how well the BDDs will do so. The performance of the pointer analysis can vary dramatically depending on this one factor. There are BDD optimizations that we have not yet explored, and these may improve performance; these include the re-arrangement of the BDD variable ordering, the use of *don't care* values in the BDD, and other formulations of BDDs such as Zero-Suppressed BDDs (ZBDDs). Various other BDD-based pointer analyses have benefitted from one or more of these optimizations [53, 81]

While for now the BDD versions have superior performance, there is still much that can be done to improve the bitmap versions. Memory is the critical factor, and most of the memory consumption comes from the local points-to graphs. Even after applying the local points-to graph equivalence optimization, a significant number of the remaining local points-to graphs contain identical information—further efforts to identify and collapse these local graphs ahead of time could have a dramatic impact on memory consumption. For example, there are several possible schemes for dynamically identifying and sharing identical bitmaps across multiple points-to graphs. In addition, by combining semi-sparse analysis with dynamically computed static single assignment form [14, 80] we could greatly reduce the sizes of the local points-to graphs. We can decrease the cost of evaluating the transfer functions using techniques such as the incremental evaluation of transfer functions [32]. We believe that there is still significant room for improvement in the bitmap version of the SSO algorithm, which we plan to explore in future work.

3.3.4.3 SSO Precision

The version of SSO used in these experiments makes use of the assumption discussed at the end of Section 3.3.2.1, i.e., that callee functions do not modify address-taken pointer information accessible by their callers. This assumption increases the effectiveness of the optimizations (see Figures 3.7 and 3.8 for a comparison), but potentially sacrifices some precision. To test how much precision is lost we compute the thru-deref metric for SSO both with and without this assumption. The thru-deref metric examines each LOAD and STORE in the program and averages the points-to set sizes of the dereferenced variables, weighted by the number of times each variable is dereferenced—the larger the value, the less precise the pointer analysis.

We find that our benchmarks do not suffer a significant precision loss by making this assumption; on average the thru-deref metric increased by 0.1%, with a maximum increase of 0.2%.

3.4 Staged Analysis

While semi-sparse analysis represents an order of magnitude improvement over the previous state-of-the-art, we can do even better. Semi-sparse analysis still analyzes address-taken variables in a non-sparse manner; by using sparse analysis for all the variables, both top-level and address-taken, we can significantly increase the scalability of the flow-sensitive analysis. This section describes a technique that meets this goal called *staged flow-sensitive pointer analysis*.

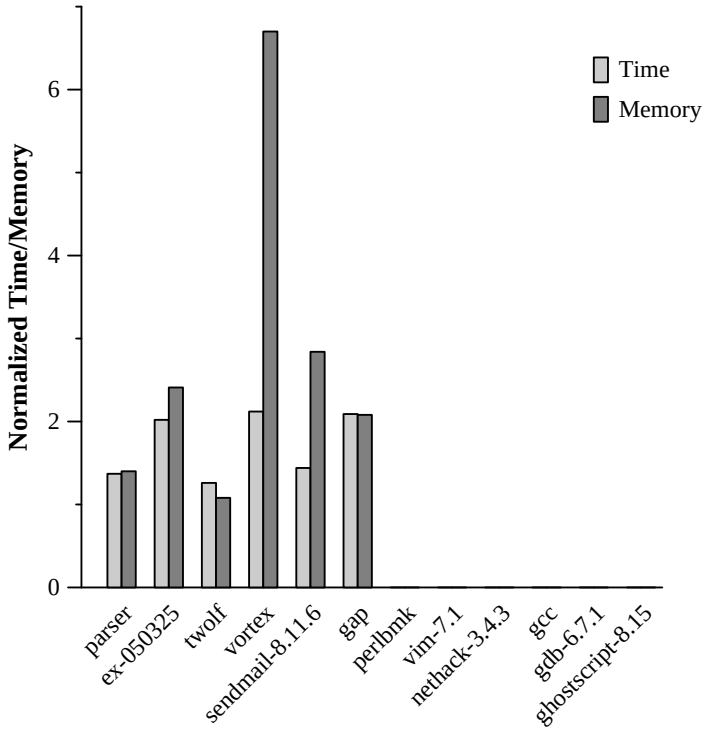


Figure 3.7: Analysis time and memory usage (normalized to our baseline) for the bitmap version of SSO *without* the assumption on CALLs versus SSO *with* the assumption—i.e., $\text{SSO}_{without} / \text{SSO}_{with}$.

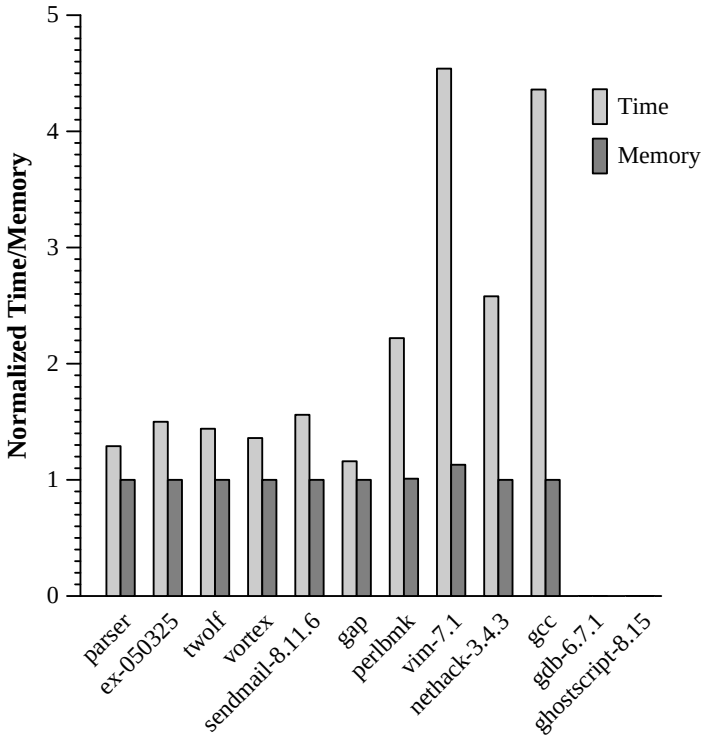


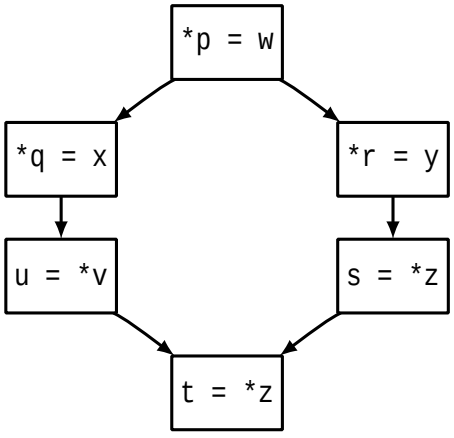
Figure 3.8: Analysis time and memory usage (normalized to our baseline) for the BDD version of SSO *without* the assumption on CALLs versus SSO *with* the assumption—i.e., $\text{SSO}_{without}/\text{SSO}_{with}$.

3.4.1 Staging the Analysis

The reason semi-sparse analysis treats address-taken variables in a non-sparse manner is that, without pointer information, the analysis cannot determine where address-taken variables are defined or used. The essential idea of our new algorithm is to enable sparse flow-sensitive pointer analysis for *all* variables in the program by staging the pointer analysis. We first employ an auxiliary, flow-insensitive pointer analysis to compute conservative def-use information for the address-taken variables of a program; we then use that information to increase the sparseness of the primary, flow-sensitive pointer analysis, thereby greatly increasing its efficiency.

3.4.1.1 Auxiliary Pointer Analysis

Any flow-insensitive pointer analysis can be used for the auxiliary analysis. There are many to choose from, ranging from the simplest address-taken analysis (which reports that any pointer can point to any variable whose address has been taken), to Steensgaard's analysis [78], to Das' One-Level Flow [25], to inclusion-based (i.e., Andersen-style) analysis, which is the most precise of all these analyses. In choosing an auxiliary pointer analysis, there are two important considerations: (1) how scalable the auxiliary analysis is, and (2) how effective its results are for optimizing the primary, flow-sensitive analysis. The more precise the auxiliary analysis, the more sparse the primary analysis will be; for this reason, together with our results on making inclusion-based (flow- and context-insensitve) analysis extremely scalable (see Chapter 2), we believe that inclusion-based analysis is the best choice. Henceforth, we will designate the chosen auxiliary pointer analysis as AUX.



AUX *points-to sets*
p → {a}
q → {b,c,d,e,f}
v → {e,f}
r → {a,b,d}
z → {a,b,c,d}

Figure 3.9: Example CFG, along with a subset of the points-to sets computed by AUX.

3.4.1.2 Sparse Flow-Sensitive Pointer Analysis

The primary data structure that we use for the sparse flow-sensitive pointer analysis is a def-use graph (DUG). The DUG contains a node for each statement in the program, and its edges represent def-use chains—if a variable is defined in node *x* and used in node *y*, there is a directed edge from *x* to *y*. The def-use edges for top-level variables are trivial to determine from inspection of the program; the def-use edges for address-taken variables require AUX to compute. This section describes how we compute these def-use edges, as well as how we maintain the precision of the flow-sensitive analysis while using flow-insensitive def-use information.

The first step is to use the results of AUX to convert the address-taken variables into SSA form. We annotate the LOADs and STOREs using χ and μ functions as described in Section 3.2.1, then convert the program to SSA form using any standard SSA algorithm [3, 8, 23, 24]. Figure 3.9 shows a small example program along with a subset of the pointer information discovered by AUX. Figure 3.10 shows the same example program, annotated with χ and μ functions and translated into SSA.

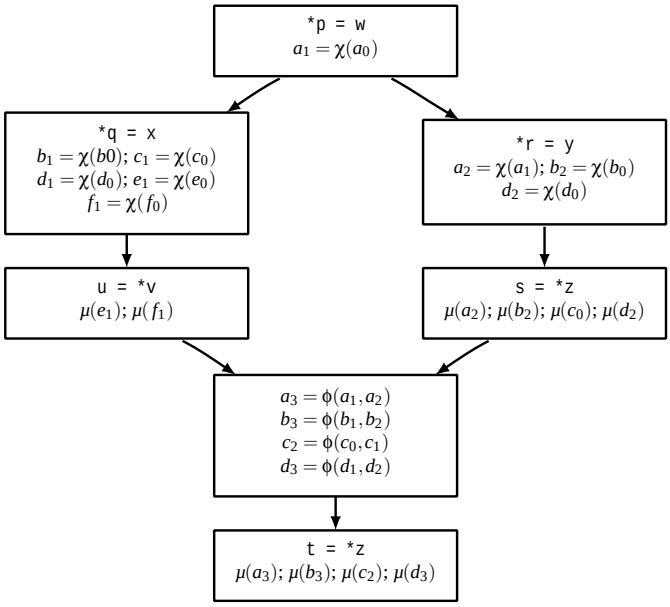


Figure 3.10: The SSA information for Figure 3.9.

Note that the def-use information revealed by the χ and μ functions and SSA form is conservative with respect to the more precise flow-sensitive information that will be computed by the primary analysis. In particular, there are three possibilities that must be addressed for a STORE `*x = y` that is annotated with $v_m = \chi(v_n)$:

1. `x` may not point to v in the flow-sensitive results. In this case, the analysis should interpret the STORE based solely on the χ function; in other words, v_m should be a copy of v_n and not incorporate `y` at all.

2. `x` may point only to v in the flow-sensitive results. In this case, the analysis can strongly update the points-to information for v; in other words, v_m should be a copy of `y` and not incorporate v_n at all.

3. `x` may point to v as well as other variables in the flow-sensitive results. In this

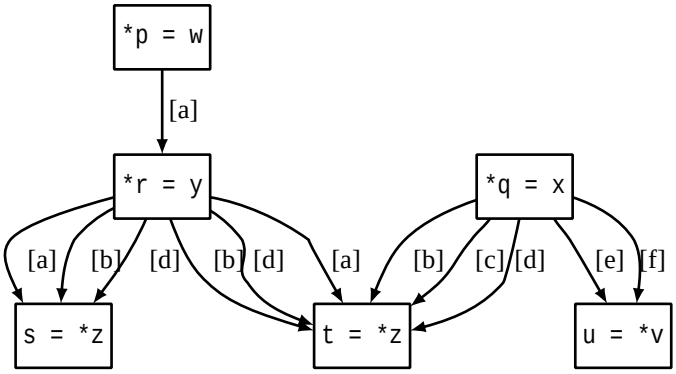


Figure 3.11: The def-use graph for Figures 3.9 and 3.10; each edge is labeled with the variables used by the destination.

case, the analysis must weakly update the points-to information for v; in other words, v_m should incorporate the points-to information from both v_n and y.

By using the SSA form to fill in the def-use graph DUG, we can accommodate all of these possibilities. For each STORE annotated with a function $v_m = \chi(v_n)$, we create a def-use edge to every statement that uses v_m as the argument of a χ, μ, or ϕ function. We label each def-use edge for an address-taken variable with that variable, so the analysis can determine along which edge to propagate a given variable's points-to information. Figure 3.11 shows the example program from Figure 3.9 converted into a def-use graph based on the SSA information from Figure 3.10.

The flow-sensitive analysis propagates to a STORE points-to information for all variables that may be defined by that STORE. When the STORE is evaluated, each variable defined by the STORE has its points-to information modified in the STORE's local points-to graph, using a strong or weak update as appropriate. The points-to information for all potentially-defined variables is then propagated along the appropriate def-use edges from the STORE, regardless of whether the STORE actually defined the variable or not.

Theorem 5 (The Analysis is Correct). Every definition of a variable reaches its corresponding uses, and the analysis computes precise flow-sensitive pointer information.

Proof. We prove the theorem in two parts:

Every def reaches its corresponding uses. Points-to information flows along the def-use chains in DUG computed by AUX. Since AUX computes an over-approximation of the information computed by the flow-sensitive analysis, the def-use chains in DUG are a superset of the def-use chains that would be computed by the flow-sensitive analysis. Therefore all defs must reach their corresponding uses.

The sparse analysis is precise. The three STORE possibilities listed above must be correctly handled by the sparse analysis. The key insight required to prove that the analysis correctly handles each possibility is that the points-to information at each DUG node increases monotonically—once a pointer contains a variable *v* in its points-to set at node *n*, that pointer will always contain *v* at node *n*. This fact constrains the transitions that each STORE can make among the three possibilities.

Suppose we have a STORE `*x = y`. First, we note that the STORE is not processed if `x` is NULL—either we will revisit this node when `x` is updated, or the program will never execute past this point (because it will be dereferencing a null pointer). Therefore if we're visiting the STORE, then `x` must point to something. The monotonicity property constrains the transitions that the analysis may take among the three possibilities for this STORE: the analysis may transition from (1) or (2) to (3), and from (1) to (2), but it can never transition from (2) to (1) and never from (3) to either (1) or (2).

More concretely, suppose that `x` does not point to *v* when the STORE is visited. Then the analysis will propagate the old value of *v* past this node. Later

in the analysis, x may be updated to point to *v*; if so, the STORE *must* be a weak update (possibility 3) because x already points to some variable other than *v* at this point in the program and it cannot change that fact. So the analysis will update *v* with both the old value of *v and* the value of y, which is a superset of the value it propagated at the last visit (the old value of *v*). Similar reasoning shows that if the STORE is originally a strong update (possibility 2) and later becomes a weak update, the analysis still operates correctly. □

3.4.1.3 Access Equivalence

A difficulty that immediately arises when using the technique described above is the sheer number of def-use edges that may be required. Each STORE can define thousands of variables, based on the dereferenced variable's points-to set size, and each variable can be defined dozens or hundreds of times—in large benchmarks, hundreds of millions of def-use edges may be created, far too many to enable a scalable analysis. To combat this problem, we introduce the notion of *access equivalence*, which will enable us to represent the same information in a much more compact fashion.

Two address-taken variables *x* and *y* are access equivalent if whenever one is accessed by a LOAD or STORE instruction, so is the other; in other words, for all variables *v* such that *v* is dereferenced in a LOAD or STORE, $x \in \textit{points-to}(v) \Leftrightarrow y \in \textit{points-to}(v)$. This notion of equivalence is similar, but not identical to the notion of *location equivalence* described in Section 2.4.2. The difference is that location equivalence examines *all* pointers in a program to determine whether two variables are equivalent, whereas access equivalence only looks at pointers dereferenced in a LOAD or STORE; two variables may be access equivalent without being location equivalent (but not vice-versa).

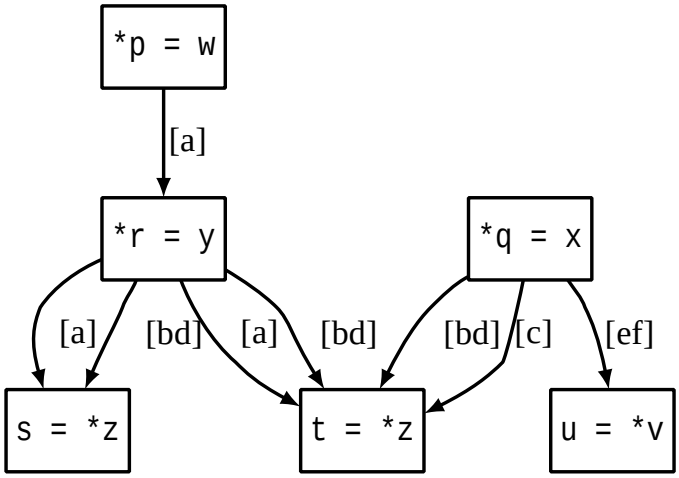


Figure 3.12: The def-use graph of Figure 3.11 after applying access equivalence.

The advantage of access equivalence is that all access-equivalent variables will have identical def-use chains computed by the SSA algorithm. By definition, any STORE that defines one variable must also define all access-equivalent variables, and similarly any LOAD that uses one variable must also use all access-equivalent variables.

To determine access equivalence using AUX, we must determine which variables are accessed by the same set of LOADs and STOREs. Let AE be a map from address-taken variables to sets of instructions. For each LOAD or STORE instruction I and for each variable v accessed by I, $AE(v)$ includes I. Once all instructions have been processed, any two variables x and y are access-equivalent if $AE(x) = AE(y)$. This process takes $O(I \cdot V)$ time, where I is the number of LOAD/STORE instructions and V is the number of address-taken variables.

For Figure 3.9, the access equivalences are: $\{a\}, \{b,d\}, \{c\}, \{e,f\}$. Figure 3.12 shows the same def-use graph as Figure 3.11 except with edges for access-equivalent variables collapsed into a single edge.

It is important to note that the access equivalences are computed using AUX, and therefore are conservative with respect to the actual access equivalences using the flow-sensitive pointer analysis. For this reason, while edges are labeled using access equivalences, the points-to graphs at each node use the actual variables. The def-use edges are now labeled with the access equivalence partition each edge represents, instead of being labeled with individual variables; when propagating a variable's points-to information across the def-use edges, the information is only propagated across edges labeled with the specific partition that variable belongs to.

3.4.1.4 Interprocedural Analysis

There are two possible approaches for extending the analysis described above to an interprocedural analysis. The first option is to compute sparseness separately for each function, treating a function call as a definition of all variables defined by the callee and as a use of all variables used by the callee. The downside of this approach is variable def-use chains can span a number of functions; treating each function call between the definition and the use as a collection point can adversely affect the sparseness of the analysis.

The second option, which we use, is to compute the sparseness for the entire program at once, directly connecting variable definitions and uses even across function boundaries. An important consideration for this approach is how to handle indirect calls via function pointers. Some of the def-use chains that span multiple functions may be dependent on the resolution of indirect calls. The technique outlined earlier does not compensate for this problem—it assumes that the def-use chains are only dependent on the points-to sets of the pointers used by an instruction, without taking into account any additional dependencies on the points-to sets of unrelated function pointers. In other words, this technique may lose precision if

the call-graph computed by AUX over-approximates the call-graph computed by a flow-sensitive pointer analysis.

There are two possible solutions to this problem. The easiest is simply to assume the AUX computes a precise call-graph, i.e., the same call-graph the flow-sensitive pointer analysis would compute. If AUX is fairly precise (e.g., an inclusion-based analysis), this is a good assumption to make—Milanova et al. [56] show that precise call-graphs can be constructed using only flow-insensitive pointer analysis. We use an inclusion-based analysis for AUX, and hence this is the solution we use for our work.

If this assumption is not desirable, then the technique must be adjusted to account for the extra dependencies. Each def-use chain that crosses a function boundary and depends on the resolution of an indirect call is annotated with the $\langle function\ pointer, target\ function \rangle$ pair that it depends on. Pointer information is not propagated across this def-use edge unless the appropriate target has been computed to be part of the function pointer's points-to set.

3.4.2 The Final Algorithm

Putting everything together, we arrive at the final algorithm for sparse flow-sensitive pointer analysis. We begin with a series of preprocessing steps prior to the analysis itself:

1. Run AUX to compute conservative def-use information for the program being analyzed.

2. Use the results of AUX to compute the interprocedural control-flow graph of the program, including resolving indirect calls to their potential targets. All function calls are then translated into a set of COPY instructions to represent

parameter assignments, and similarly function returns are also translated into COPY instructions.

3. Compute exact SSA information for all top-level variables.

4. Partition the address-taken variables into access equivalence classes as described in Section 3.4.1.3.

5. For each partition P, use the results of AUX to:

   - Label each STORE that may modify a variable in P with a function $P = \chi(P)$.
   - Label each LOAD that may access a variable in P with a function $\mu(P)$.

6. Compute SSA form for the partitions, using any of many available methods (e.g., [3, 8, 23, 24]).

7. Construct the def-use graph by creating a node for each pointer-related instruction and each ϕ function created by step 6, then:

   - For each ALLOC, COPY, and LOAD node N, add an unlabeled edge from N to every other node that uses the variable defined by N. (Note that because of step 3, nodes of these types each define a unique variable; the COPY nodes include the ϕ functions computed by step 3.)
   - For each STORE node N that has a χ function defining a partition variable P_n, add an edge from N to every node that uses P_n (either in a ϕ, χ or μ function), labeled by the partition P.
   - For each ϕ node N that defines a partition variable P_n, create an unlabeled edge to every node that uses P_n.

Once the preprocessing is complete, the sparse analysis itself can begin. The analysis uses the following data structures:

- a node worklist *Worklist* that is initialized to contain all ALLOC nodes.
- a global points-to graph *PG* that holds the points-to sets for all top-level variables. Let $\mathcal{P}_{top}(v)$ be the points-to set for top-level variable v.
- a points-to graph IN_k for every LOAD and ϕ node k to hold the pointer information for all address-taken variables that may be accessed by that node. Let $\mathcal{P}_k(v)$ be the points-to set for address-taken variable v contained in IN_k.
- two points-to graphs for every STORE node k to hold the pointer information for all address-taken variables that may be defined by that node: IN_k for the incoming pointer information and OUT_k for the outgoing pointer information. Let $\mathcal{P}_k(v)$ be the points-to set of address-taken variable v in IN_k.
- a map $part(v)$ that for each address-taken variable v returns the variable partition to which that v belongs.

The main body of the algorithm is listed in Algorithm 15. The loop iteratively selects a node from the worklist and processes it, which may add new nodes to the worklist. It continues until the worklist is empty, at which point the analysis is complete. Each different type of node is processed as listed in Algorithms 16–20. The $\hookleftarrow$ operator represents set update, $\rightarrow$ represents an unlabeled edge in the def-use graph, and $\xrightarrow{x}$ represents an edge labeled with x.

Algorithm 15 Main body of the semi flow-sensitive pointer analysis algorithm.

Require: $DEF/USE = \langle N, E \rangle$
  while *Worklist* is not empty **do**
    $k =$SELECT$(Worklist)$
    switch *typeof*(k)**:**
      case ALLOC: processAlloc(k)
      case COPY: processCopy(k)
      case LOAD: processLoad(k)
      case STORE: processStore(k)
      case ϕ: processPhi(k)

Algorithm 16 processAlloc(k) : [x = ALLOC$_i$]

  $PG \hookleftarrow \{\mathsf{x} \rightarrow \text{ALLOC}_i\}$
  if *PG* changed **then**
    $Worklist \hookleftarrow \{\ n \mid k \rightarrow n \in E\}$

Algorithm 17 processCopy(k) : [x = y z ...]

  for all $v \in$ right-hand side **do**
    $PG \hookleftarrow \{\mathsf{x} \rightarrow \mathcal{P}_{top}(v)\}$
  if *PG* changed **then**
    $Worklist \hookleftarrow \{\ n \mid k \rightarrow n \in E\}$

Algorithm 18 processLoad(k) : [x = *y]

  $PG \hookleftarrow \{\mathsf{x} \rightarrow \mathcal{P}_k(\mathcal{P}_{top}(\mathsf{y}))\}$
  if *PG* changed **then**
    $Worklist \hookleftarrow \{\ n \mid k \rightarrow n \in E\}$

Algorithm 19 processStore(k) : [*x = y]

if $\mathcal{P}_{top}(\mathsf{x})$ represents a single memory location **then**
  // *strong update*
  $\text{OUT}_k \hookleftarrow (\text{IN}_k \setminus \mathcal{P}_{top}(\mathsf{x})) \cup \{\mathcal{P}_{top}(\mathsf{x}) \rightarrow \mathcal{P}_{top}(\mathsf{y})\}$
else // *weak update*
  $\text{OUT}_k \hookleftarrow \text{IN}_k \cup \{\mathcal{P}_{top}(\mathsf{x}) \rightarrow \mathcal{P}_{top}(\mathsf{y})\}$
for all $\{n \in N, p \in P \mid k \xrightarrow{p} n \in E\}$ **do**
  for all $\{v \in \text{OUT}_k \mid part(v) = p\}$ **do**
    $IN_n(v) \hookleftarrow OUT_k(v)$
    if IN_n changed **then**
      $Worklist \hookleftarrow \{n\}$

Algorithm 20 processPhi(k)

for all $\{n \in N \mid k \rightarrow n \in E\}$ **do**
  $IN_n \hookleftarrow IN_k$
  if IN_n changed **then**
    $Worklist \hookleftarrow \{n\}$

3.4.2.1 Further Optimization

In addition to the techniques described in this section, we can also use the same two optimizations described in the previous section on semi-sparse analysis, namely *Top-Level Pointer Equivalence* and *Local Points-to Graph Equivalence*. We employ both optimizations in the following experimental evaluation.

3.4.3 Evaluation

To evaluate our new technique, we compare it against our earlier work on flow-sensitive pointer analysis, called SSO(see Section 3.3), which is the most scalable algorithm available. SSO analyzes benchmarks with up to approximately 344K lines of code (LOC), an order of magnitude greater than allowed by the previous state-of-the-art, and it is almost 200× faster than the previous state-of-the-art. We

use SSO as the baseline for comparison with our new technique, which we refer to as SFS. SFS uses inclusion-based (i.e., Andersen-style) analysis for AUX. SSO, SFS, and AUX are all field-sensitive—each field of a struct is treated as a separate variable.

We implement both SSO and SFS in the LLVM compiler infrastructure [52] and use BDDs to store points-to relations. We emphasize that neither technique is a symbolic analysis (such as the various symbolic pointer analyses [7, 81, 85–87])—instead, we only use BDDs to compactly represent points-to sets; we could swap in other data structures for this purpose without changing the rest of the analysis. We make use of the BuDDy BDD library [54]. The analyses are written in C++ and handle all aspects of the C language except for varargs.

Table 3.5 describes the benchmarks for our experiments. Six of the benchmarks are taken from SPECINT 2000 (the largest six applications from that suite) and the rest from various open-source applications. These benchmarks include all the benchmarks from the evaluation of semi-sparse analysis (Section 3.3.4), plus the benchmarks svn (a source control system), gimp (an image manipulation program), and tshark (a wireless network analyzer). We do not include the Linux or Wine benchmarks used for the evaluation of inclusion-based analysis because of difficulties compiling those programs with the LLVM infrastructure (note that the inclusion-based analysis evaluation in Sections 2.3.3 and 2.4.3 used a different infrastructure than LLVM). Function calls to external code are summarized using hand-crafted function stubs. The experiments are run on a 2.66 GHz 32-bit processor with 4GB of addressable memory, except for our largest benchmark, tshark, which uses more than 4GB of memory—that benchmark is run on a 1.60 GHz 64-bit processor with 100GB of memory. Note that the experimental machines are different from that used for Section 3.3.4 and therefore we get different results than

Name	LOC	Statements	TL Vars	AT Vars
197.parser	11K	18K	7.6K	1.9K
300.twolf	20K	37K	12.4K	4.8K
ex	34K	37K	8.8K	2.3K
255.vortex	67K	47K	15.3K	5.9K
254.gap	71K	99K	39.8K	8.2K
sendmail	74K	54K	20.2K	28.5K
253.perlbmk	82K	118K	48.8K	4.1K
nethack	167K	298K	79.0K	15.1K
python	185K	162K	70.7K	21.9K
176.gcc	222K	258K	108.0K	12.9K
vim	268K	249K	74.8K	168.0K
pine	342K	426K	206.0K	404.0K
svn	344K	158K	83.5K	23.8K
ghostscript	354K	388K	164.0K	350.0K
gimp	877K	929K	408.0K	146.0K
tshark	1,946K	1,045K	914.0K	641.0K

Table 3.5: Benchmarks: **LOC** reports the number of lines of code, **Statements** reports the number of statements in the LLVM IR, **TL Vars** reports the number of top-level variables, and **AT Vars** reports the number of address-taken variables. The benchmarks are divided into small (less than 100K LOC), mid-sized (between 100K–400K LOC), and large (800K LOC and greater).

that section for SSO.

3.4.3.1 Performance Results

Table 3.6 gives the analysis time and memory consumption of the various algorithms. These numbers include the time to build the data structures, apply the optimizations, and compute the pointer analysis. The times for SFS are additionally broken down into the three main stages of the analysis: the auxiliary flow-insensitive pointer analysis, the preparation stage that computes sparseness, and the

Name	SSO		SFS				
	Time	Mem	Prelim	Prep	Solve	Time	Mem
parser	0.41	138	0.29	0.07	0.008	0.37	275
twolf	0.23	140	0.34	0.07	0.004	0.41	281
ex	0.35	141	0.29	0.10	0.008	0.40	277
vortex	0.60	144	0.45	0.14	0.028	0.62	285
gap	1.28	155	0.94	0.33	0.016	1.29	307
sendmail	1.21	147	0.70	0.27	0.032	1.00	301
perlbmk	2.30	158	1.05	0.50	0.020	1.57	312
nethack	3.16	197	1.72	0.82	0.096	2.64	349
python	120.16	346	4.04	2.02	0.564	6.62	404
gcc	3.74	189	2.00	1.42	0.040	3.46	370
vim	61.85	238	2.93	2.44	0.160	5.53	436
pine	347.53	640	13.42	21.25	47.330	82.00	876
svn	185.10	233	5.40	5.07	0.216	10.69	418
ghostscript	OOT	—	42.98	86.13	1787.184	1916.29	2359
gimp	OOT	—	90.59	105.87	1025.824	1222.28	3273
tshark	OOT	—	232.54	219.83	376.096	828.47	6378

Table 3.6: Performance: time (in seconds) and memory (in megabytes) of the analyses. OOT means the analysis ran out of time (exceeded a 1 hour time limit). SFS is broken down into the main stages of the analysis: the auxiliary pointer analysis, the preparation stage that computes sparseness, and the actual time to solve.

solver stage.

The premise of SFS—that approximating a sparse analysis by using an auxiliary pointer analysis to conservatively compute def-use information—is clearly borne out. For the smaller benchmarks, those less than 100K LOC, the advantage is less clear; sometimes SSO is faster, sometimes SFS is faster. For the benchmarks with less than 100K LOC, SFS is on average 1.03× faster than SSO. For the mid-sized benchmarks, those with between 100K LOC and 400K LOC, SFS has a more distinct advantage; it is on average 5.5× faster than than SSO for the six bench-

marks that both algorithms complete. In addition, SFS successfully analyzes three benchmarks, each in less than $\frac{1}{2}$ hour, that SSO cannot analyze within an hour.

The one area where SSO has a clear advantage is memory consumption. SFS has not been tuned with respect to memory consumption, and we believe its memory footprint can be significantly reduced. As a sidenote, keep in mind that tshark is evaluated using a 64-bit machine, as opposed to the 32-bit machine used for the other benchmarks, so its memory consumption can't be directly compared with the others because the 64-bit machine inflates the memory footprint compared to a 32-bit machine.

3.4.3.2 Performance Discussion

There are several observations about the SFS results that may seem surprising. First, the solve times for SFS are sometimes smaller than the AUX times. Keep in mind that the AUX column includes the time needed for AUX to generate constraints, optimize those constraints, solve them, then do some post-processing on the results to prepare them for the SFS solver. On the other hand, the solve times only include the time taken for SFS to actually compute an answer given the def-use graph.

We also see that the analysis times can vary quite widely, even for benchmarks that are close in size. Some smaller benchmarks take significantly longer than larger benchmarks. The analysis time for a benchmark depends on a number of factors besides the raw input size: the points-to set sizes involved; the characteristics of the def-use graph, which determines how widely pointer information is propagated; how the worklist algorithm interacts with the analysis; and so forth. It is extremely difficult to predict analysis times without knowing such information, which can only be gathered by actually performing the analysis.

Finally, the prep time for SFS, which includes the time to compute SSA information using the AUX results and the time to optimize the analysis using Top-level Pointer Equivalence and Local Points-to Graph Equivalence, takes a significant portion of the total time for SFS. While the prep stage is compute-intensive, there are several optimizations for this stage that we have not yet implemented. We believe that the times for this stage can be significantly reduced.

To better understand the results, we focus on three key aspects of SFS that contribute to its success, the analysis' sparsity, the effect of access equivalence, and the effects of local points-to graph equivalence.

The first aspect is the effect of using a sparse analysis for the address-taken variables. We measure this effect by counting, for each address-taken variable *v*, the number of edges that *v*'s points-to information can propagate across. The sparser the analysis, the fewer edges a variable's information will propagate across, and the more quickly the analysis will complete. Figure 3.13 compares for each benchmark the average number of edges a variable's information will propagate across for a non-sparse analysis versus SFS's sparse analysis. As expected, the sparse analysis propagates information across far fewer edges for every benchmark.

The second aspect is the effect of exploiting access equivalence. We use access equivalence to partition address-taken variables so that we only need def-use edges per partition, rather than per variable. Figure 3.14 compares the number of partitions versus the number of address-taken variables, and also the number of def-use edges used for partitions versus the number of edges that would be required if they were per-variable. Most of the benchmarks, and all of the larger benchmarks, show a significant reduction in the number of edges required. For the larger benchmarks, this reduction in absolute terms was from hundreds of millions of edges to millions of edges.

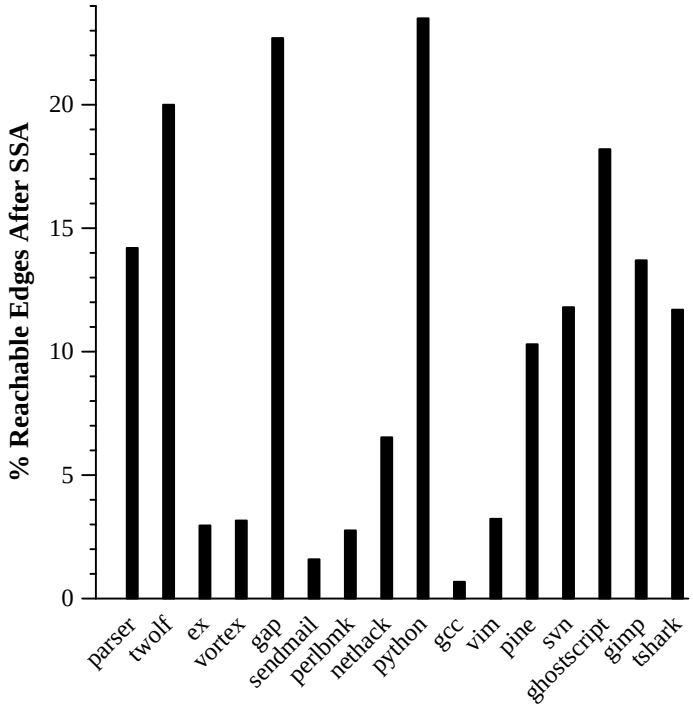


Figure 3.13: This graph reveals the sparsity of the def-use graph by giving the percentage of edges across which pointer information will be propagated in the def-use graph in relation to a non-sparse analysis using the CFG. Smaller is better: the smaller the percentage, the fewer edges a variable's pointer information will be propagated across.

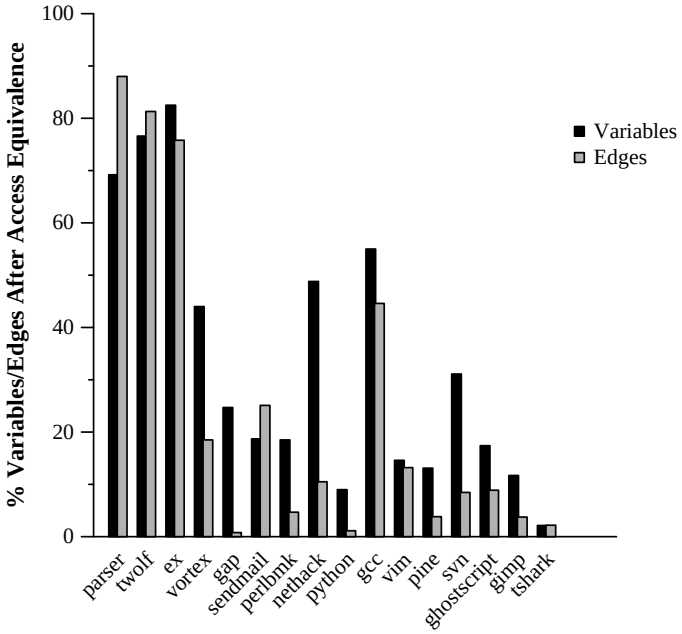


Figure 3.14: This graph shows the effectiveness of the access equivalence optimization in two ways: Vars is the remaining number of variables after replacing each variable with a representative from its access equivalence class; Edges is the number of remaining def-use edges after merging edges for variables from the same access equivalence class. Both are given as a percentage of the number of variables and def-use edges without using access equivalence. Smaller is better: the smaller the percentage, the fewer variables and edges remain in the graph.

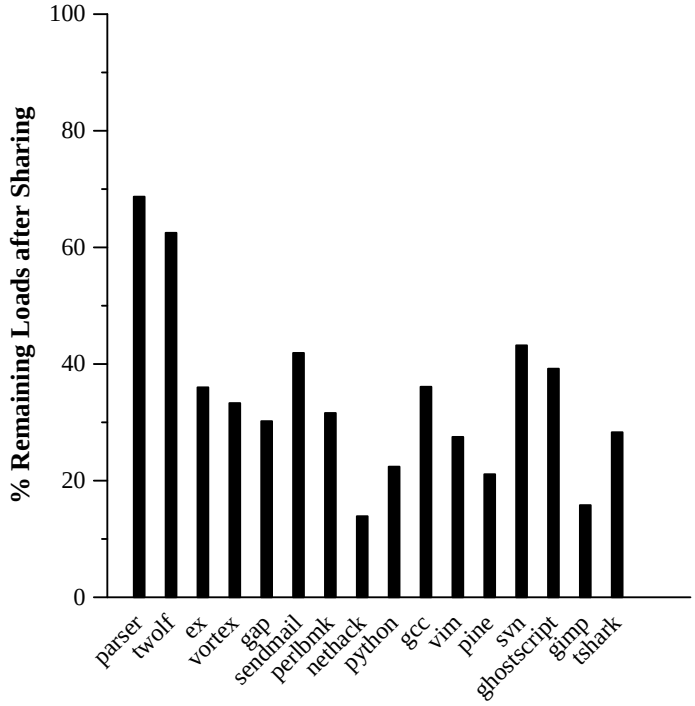


Figure 3.15: This graph shows the effectiveness of sharing points-to graphs using local points-to graph equivalence by giving the number of LOAD instructions that remain after merging nodes that share points-to graphs, as a percentage of the total number of LOADs. Smaller is better: the smaller the percentage of remaining nodes, the more sharing is being done.

The final aspect that we consider is the effect of the local points-to graph equivalence optimization for both SSO and SFS. Figure 3.15 shows the percentage of the number of nodes that remain after merging nodes that share local points-to graphs. We see that the optimization is quite effective.

3.5 Chapter Summary

In this chapter, we demonstrate how to greatly increase the scalability of flow-sensitive, context-insensitive pointer analysis. Our strategy is to exploit *sparsity* in the analysis, allowing the analysis to converge faster and use less memory.

Our first technique, semi-sparse analysis, partitions variables between *top-level* (variables that cannot be indirectly referenced via a pointer) and *address-taken* (variables that can be indirectly referenced via a pointer). The top-level variables are put into SSA form, while the address-taken variables are connected via an SEG. The semi-sparse analysis combines a sparse analysis on the top-level variables with an iterative dataflow analysis on the address-taken variables.

Our second technique, staged analysis, improves on semi-sparse analysis by transforming the address-taken variables into a conservative SSA form. This transformation allows the address-taken variables to be analyzed using a sparse analysis, rather than the iterative dataflow analysis used by semi-sparse analysis. In order to make this technique practical, we identify and exploit *access equivalence* among the address-taken variables, i.e., variables that are guaranteed to have identical def-use chains in SSA form.

Chapter 4

Formal Framework

The previous two chapters of this thesis have focused on two points in the *flow-sensitivity* dimension of precision. In the bigger picture, there are many other possible dimensions of precision for pointer analysis that can be approximated in different ways (e.g., context-sensitivity, the heap model, pointer arithmetic, etc). Previous work has explored many different points in this larger space of possible approximations, as evidenced in Hind's survey paper [43].

Different points in this space of approximations yield different trade-offs between the precision of the pointer analysis and the scalability of the analysis. This tradeoff between precision and performance is of utmost importance to those who rely on pointer analysis. Unfortunately, the current state of pointer analysis research makes it difficult for researchers to clearly communicate the trade-off associated with a given point in the space of approximations. Pointer analysis researchers tend to focus on algorithmic design, employ a multitude of algorithmic strategies (e.g., dataflow analysis [44, 50], set constraints [29, 37], type systems [25, 78], CFL reachability [77, 84]), use different terminology to refer to similar concepts, and often use informal language to describe the precision of pointer analysis approximations. These factors make it difficult for researchers to formally and empirically compare pointer analyses [43].

Researchers lack a unifying, formal specification of pointer analysis. Such a specification would make it possible to describe and compare the precision of

many different pointer analysis approximations, regardless of their implementation details. It would provide a common vocabulary for those who perform pointer analysis research and those who benefit from it.

A formal specification of pointer analysis also serves as a necessary component of a more ambitious research goal: the automatic or semi-automatic synthesis of provably correct and efficient pointer analysis algorithms. Algorithm synthesis already exists for other domains such as databases [15], planning and scheduling [65], general fixed-point computations [11], and general dataflow analysis [6]. Each technique turns a declarative description of desired results into an algorithm that computes those results; and each technique depends on the existence of a formal specification.

This chapter describes a formal framework to precisely describe a large space of possible pointer analysis approximations. This framework is capable of describing the precision of the vast majority of existing pointer analyses, regardless of how those analyses are implemented algorithmically. While the framework does not immediately lead to an efficient implementation of a formally specified approximation, we believe that it represents an important step towards the goal of declarative pointer analysis. The work described in this chapter has been previously published by Hardekopf et al. [36].

4.1 Framework Strategy

The material in this chapter is dense and full of formalisms. To help the reader navigate the chapter, we describe in this section the basic strategy of our framework in an informal manner. The framework consists of 2 essential parts: (1) a *base semantics*, and (2) three *semantic transformations*. The base semantics rep-

resents the most precise possible pointer analysis, and its complexity is exponential in the program size (it executes each path in the program independently, and there are an exponential number of program paths).

The three semantic transformations are: (1) *Induced Variable Equivalence* (IVE), (2) *Induced Trace Equivalence* (ITE), and (3) *Induced Control Flow* (ICF). The transformations can be applied to the base semantics in various combinations in order to model various approximations, reducing the precision of the base semantics. IVE forces the base semantics to treat a given set of variables as equivalent, i.e., all variables in the group must have the same pointer information. ITE forces the base semantics to treat a given set of program paths as equivalent, i.e., the results of the analysis don't depend on which path in that group is taken. ICF forces the base semantics to add additional control flow that was not present in the original program, i.e., it creates new program paths that did not previously exist.

We emphasize that the result of these transformations is not an actual, usable algorithm for computing these approximations. Instead, the set of transformations used describe a given level of precision: the algorithm designer can state that the precision of a given algorithm is identical to the precision of the base semantics with a certain combination of semantic transformations applied to it. This description makes the approximations used by the given algorithm explicit and precise.

4.2 Background

This section introduces terminology and uses the framework of dataflow analysis to describe the most common approximations made for pointer analysis. To keep the analysis decidable, all the approximations assume a finite heap model (i.e., that there is a finite bound on the amount of dynamic memory allocated) and

uninterpreted branch conditions (i.e., all branches nondeterministic). We begin with a discussion of intraprocedural analysis before extending the discussion to the interprocedural case.

4.2.1 Dataflow and Pointer Analysis

Dataflow analysis employs a lattice of dataflow facts $\mathcal{L}$, a meet operator on the lattice $\bigsqcap$, and a family of monotone transfer functions $f_i : \mathcal{L} \to \mathcal{L}$ that map lattice elements to other lattice elements. For pointer analysis, the lattice elements are elements of the powerset of possible pointer information, the meet operator is set union, and the transfer functions compute the effects of program statements on pointer information. Analysis is carried out on the *control-flow graph* (CFG), a directed graph $G = \langle N, E, s \rangle$ with a finite set of nodes (or *program points*) N, corresponding to program statements, a set of edges $E \subseteq N \times N$ corresponding to the control flow between statements, and a designated start node s such that all nodes are reachable from s. A path π in the CFG is a sequence of nodes such that there exists an edge from each node to the next node in the sequence. Each node n is associated with a transfer function f_n that computes the effects of the associated program statement. The *path semantics* for a path is the composition of the transfer functions for all the nodes contained in the path: $[\![n_1, n_2, \ldots, n_k]\!] = f_k \circ \ldots \circ f_2 \circ f_1$.

A *flow-sensitive* analysis respects the restrictions on control flow embodied by the CFG and computes a separate solution for each program point. The precise flow-sensitive solution is known as *meet-over-all-paths* (MOP):

$$\forall n \in N : \mathrm{MOP}(n) = \bigsqcap \{ [\![\pi]\!] \mid \pi \text{ is a path from } s \text{ to } n \}$$

Each program path is analyzed independently. The final solution for each program point is the meet (i.e., union) of all the pointer information relevant to that

program point. Flow-sensitive meet-over-all-paths pointer analysis (FS-MOP) is PSPACE-complete [59].

Because the precise solution is intractable, practical flow-sensitive analysis instead computes the *maximal-fixed-point* (MFP) solution:

$$\forall n \in N : \mathrm{MFP}(n) = \bigsqcap \{ f_m(\mathrm{MFP}(m)) \mid \langle m, n \rangle \in E \}$$

The analysis computes the maximal lattice element (i.e., most precise pointer information) for which the set of transfer functions converges to a fixed-point. Rather than computing solutions for a potentially unbounded number of paths, the MFP approach merges all the solutions that reach a given node along any path. For pointer analysis, the transfer functions are not distributive, so the MFP solution is an over-approximation of the MOP solution [48]. Flow-sensitive maximal-fixed-point pointer analysis (FS-MFP) is $O(n^6)$, where n is the number of program statements.

In contrast, a *flow-insensitive* analysis does not respect control flow. It assumes that any program statement can follow any other program statement and can be executed an arbitrary number of times. Equivalently, it assumes that the CFG is complete, including self-loops. A flow-insensitive analysis computes a single solution that holds for the entire program. Just as for the flow-sensitive case, the precise flow-insensitive solution is defined as the meet-over-all-paths solution of the modified CFG. Flow-insensitive meet-over-all-paths pointer analysis (FI-MOP) is NP-hard [46].

As with flow-sensitivity, there is a maximal-fixed-point flow-insensitive solution that is an over-approximation of the FI-MOP solution. Flow-insensitive maximal-fixed-point pointer analysis (FI-MFP) is $O(v^3)$, where v is the number of program variables.

4.2.2 Interprocedural analysis

There are several ways to extend intraprocedural dataflow analysis to the interprocedural case. A *context-sensitive* analysis respects the semantics of procedure calls and returns by analyzing each distinct context of a procedure independently. Without recursion, context-sensitive analysis is equivalent to inlining all procedure calls, and its complexity is exponential in the number of calls in the program. There are two general approaches to context-sensitivity that define the meaning of "distinct context" in different ways: *functional* and *call-string* [76].

The functional approach memoizes procedures. Whenever a procedure is re-analyzed using incoming pointer information that has been previously seen, the analysis re-uses the results from the previous computation. This approach distinguishes contexts dynamically during the analysis based on the incoming pointer information.

The call-string approach statically distinguishes contexts based on the string of procedure calls made to reach the procedure, i.e., the call-stack. The number of possible call-strings is infinite in the presence of recursion. The traditional response is to *k-limit* the call-string by using only the last k elements of the string to distinguish context, for some constant k.

A *context-insensitive* analysis does not respect the semantics of procedure calls and returns. It instead treats all calls and returns as goto statements. Information from multiple callers is merged before analyzing a procedure, and information passed to a procedure by one caller is passed back to all of the procedure's callers. This merging of information means that the context-insensitive solution is an over-approximation of the context-sensitive solution.

4.2.3 Other Approximations

The representation of the pointer information itself can have an impact on the precision of the analysis. The two standard representations are *alias relations* [51] and *points-to relations* [28] (the alternative *compact alias representation* is closely related to the points-to relation [17]). The points-to relation expresses the fact that a pointer may hold the address of a particular memory location (e.g., if x can hold the address of y and y can hold the address of z then $x \rightarrow y, y \rightarrow z$). The alias representation explicitly lists all pairs of pointer expressions that may resolve to the same memory location (e.g., $\langle *x, y \rangle, \langle *y, z \rangle, \langle **x, z \rangle$). In effect, the alias representation is the transitive closure of the points-to relation.

For FS-MOP and flow-insensitive analysis, these representations are equivalent, but for FS-MFP their precision is incomparable — each may be more precise than the other in different circumstances [55]. However, when using *strong updates* (explained further in Section 4.4.4) the alias relation is strictly more precise than the points-to relation for FS-MFP [55]. Our work uses the alias relation coupled with strong updates to provide maximum precision; in Section 4.5.2 we discuss the exact difference between the representations and how to use our transformations to yield precision equivalent to using points-to relations.

Two other dimensions of precision are *field sensitivity* and the *heap model*. Field-sensitivity specifies how individual fields of a struct are distinguished. The heap model specifies how the (conceptually) infinite-size heap is abstracted into a finite number of abstract memory locations. For simplicity our framework does not directly address field-sensitivity, but Section 4.5.3 briefly describes how our framework could handle it. The heap model is addressed in Section 4.7.2.

4.3 Related Work

Bruns and Chandra have previously developed a theoretical model to describe points-to analysis approximations, using it to explore the relationships among some of the common intraprocedural approximations [9]. They introduce four coarse-grained semantic transformations that compose to specify solutions equivalent to FS-MOP, FS-MFP, FI-MOP, and FI-MFP. They introduce an additional transformation that suggests a more efficient FI-MFP algorithm. Our work is inspired by their effort, but we seek to create a framework that is both simpler and at the same time more expressive than the one they describe. We employ three semantic transformations that specify the same approximations as Bruns and Chandra plus many more, including heap models and interprocedural approximations. Our transformations are designed to tune the precision of an approximation, rather than the efficiency of any one algorithm. Our model uses aliasing rather than points-to relations in order to create a more general framework.

Milanova and Ryder describe a practical framework for FI-MFP pointer analysis that uses annotated inclusion constraints [57]. By varying the annotations, their framework can specify a wide range of inclusion-based flow-insensitive analyses, from context-insensitive to both the call-string and functional approaches to context-sensitivity.

Grove et al. describe a lattice-theoretic model of context-sensitive call-graphs which they use to specify different families of call-graph construction algorithms, leading to a parameterized framework for interprocedural context-sensitive analysis [33]. Their framework does not directly address pointer analysis or flow-sensitivity.

Deutsch [26] and Sagiv et al. [71] present parametric frameworks that describe broad spectra of *shape analysis* approximations. The boundary between

shape analysis and pointer analysis is nebulous. For the purposes of this dissertation, we will define pointer analysis as an approximation of stack-allocated pointers and shape analysis as an approximation of heap-allocated (and often cyclic) data. This paper's focus is pointer analysis approximations. Nevertheless, our model can describe a subset of the shape analysis approximations described by Deutsch and Sagiv et al. Section 4.7.2 discusses these issues in more detail.

4.4 Intraprocedural Reference Model

The intraprocedural reference model provides the base semantics for our framework. The reference model computes the FS-MOP solution for a program that contains a single procedure. We give an informal overview of the reference model followed by a formal definition of the model's syntax and semantics. Section 4.6 discusses how to extend this model to apply to programs that contain multiple procedures.

Our framework is a form of *abstract interpretation* [21], where a pointer analysis approximation abstracts away information from the base reference model. In this view, our base reference model acts as the concrete semantics and computes the precise FS-MOP solution of alias relations that occur in a program. Abstract interpretation traditionally involves separate concrete and abstract domains, but our framework uses a single domain that can express degrees of abstraction. It is in effect an abstract domain that contains the concrete domain as a special case. We define a single semantics that works for both concrete and abstract computations and vary the degree of abstraction as desired.

4.4.1 Overview

A program in the reference model consists of a control-flow graph for a single procedure with branches and assignment statements. The branches do not have conditions. Instead, we follow common practice in replacing branch conditions with nondeterministic gotos in order to guarantee decidability.

We define a *trace-based* semantics [73]. The semantics is nondeterministic and generates a program's *computation tree*. This tree represents all the program's possible execution paths. Each path in the tree is a *trace* that corresponds to one possible execution of the program. Each node in the trace contains information about the program's store (i.e., the computed pointer information) at a given statement.

The store is modeled as a set of alias relations. If the store contains the relation $(*^i x, y)$ then the expression $*^i x$ (i.e., the variable x dereferenced i times) and the variable y are aliases. The reference model semantics ensures that the store remains closed under reachability. The semantics also ensures that the store remains finite, even in the presence of cycles among the alias relations.

The non-deterministic semantics may generate computation trees of infinite size; but because the store is finite, an infinite computation tree must be regular — every trace contains a repeating node. This repetition ensures that the computation can safely terminate once all its traces reach a repeated node.

We now provide formal definitions for the syntax and semantics of the intraprocedural reference model. Section 4.5 defines three transformations that may be combined to describe a particular pointer analysis approximation.

$$
\begin{array}{rcl}
 & n \in \mathbb{N} \quad x \in \mathit{Variable} \quad \rho \in \mathit{ProgPoint} & \\
e \in \mathit{PtrExpr} & ::= & \&x \mid *^n x \\
s \in \mathit{Stmt} & ::= & *^n x := e \mid \textbf{skip} \\
\mathit{Pr} \in \mathit{Program} & : & \mathit{ProgPoint} \to \mathit{Stmt} \times \overline{\mathit{ProgPoint}}
\end{array}
$$

Figure 4.1: An unstructured pointer language without procedures.

4.4.2 Syntax

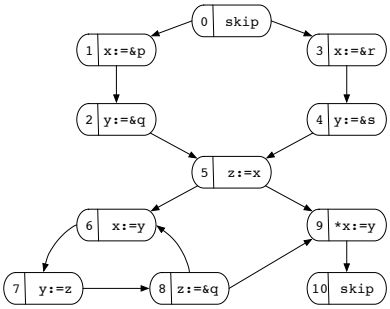
The syntax of expressions and statements is given in Figure 4.1. A pointer expression can take the address of a variable ($\&x$) or dereference a variable ($*^n x$) an arbitrary number of times, where $*^0 x \equiv x$. The only statements are assignment and **skip**. The statements in a program are labeled by a finite set of *program points*.

A *program* is a control-flow graph, which is defined as a mapping $Pr(\rho)$ from program points to a pair $\langle \mathit{statement}, \mathit{goto\ set} \rangle$. At program point ρ with $Pr(\rho) = \langle s, \mathcal{P} \rangle$ the program executes statement s and then branches nondeterministically to one of the program points in the goto set $\mathcal{P}$. The distinguished program point ρ_0 refers to the program's unique entry point. A program also has a unique exit point that maps to the pair $\langle \textbf{skip}, \emptyset \rangle$.

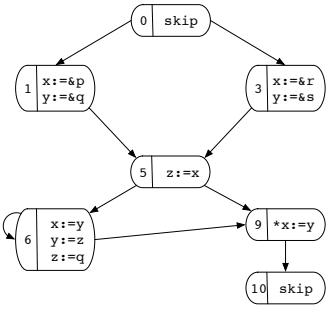
Figure 4.2 gives an example program (a) and its corresponding control-flow graph (b). This program represents a structured program with a conditional (statements 0–4) and a loop (statements 6–8). The program's entry point is 0 and its exit point is 10. For brevity, some of our examples will use a basic-block form (c) of the program.

ρ	*statement*	$\mathcal{P}$
$\rho_0 = 0$	**skip**	$\{1,3\}$
1	x := &p	$\{2\}$
2	y := &q	$\{5\}$
3	x := &r	$\{4\}$
4	y := &s	$\{5\}$
5	z := x	$\{6,9\}$
6	x := y	$\{7\}$
7	y := z	$\{8\}$
8	z := &q	$\{6,9\}$
9	$*$x := y	$\{10\}$
10	**skip**	$\emptyset$

(a) Program definition.



(b) Control-flow graph.



(c) Basic-block control-flow graph.

Figure 4.2: The definition, corresponding control-flow graph, and basic-block graph for a program with one conditional and one loop. The basic-block graph is a more compact representation that helps condense the presentation of some examples in this chapter. Each basic block is labeled with the program point of the first statement in the block.

4.4.3 Semantic Domain

The semantic domain consists of computation trees where each path in the tree is a trace of one possible program execution. Each node in the trace describes the alias relations at a particular program point. We now provide formal definitions for these domains. We also provide order relations, which are useful for proving soundness and termination. Section 4.4.4 defines how the program semantics manipulate these domains.

Stores. A store is a set of relations:

$$\sigma \in \mathit{Store} : \overline{\mathbb{N} \times \mathit{Variable} \times \mathit{Variable}}$$

If the store contains the relation $(*^i x, y)$, then the expression $*^i x$ and the variable y are aliases. *Store*'s order relation $\sqsubseteq$ is the subset relation ($\subseteq$), and its join operator $\sqcup$ is set union ($\cup$).[1] We sometimes want to refer to all the pointer variables in a store, which we denote by the function *dom* (i.e., *domain*):

$$\mathit{dom}(\sigma) = \{x \mid (*^i x, y) \in \sigma\}$$

Figure 4.3 defines three operations on stores: *lookup*, *insert*, and *delete*. The lookup operation $\sigma^i[x]$ yields the set $\{y \mid (*^i x, y) \in \sigma\}$, where $\sigma^0[x] = \{x\}$.

The insert operation $[x \overset{i}{\mapsto} y]\sigma$ adds the relation $(*^i x, y)$ to the store. The store must remain closed under reachability, so the insert operation also adds all transitive relations that result from adding the specified relation. If the newly added relation induces a cycle in the alias relations, then the new store will be infinite.

[1] The reference model follows the convention used in abstract interpretation, where the least element is the most precise and the merge operation is join. In Section 4.2, we followed the convention used in dataflow analysis where the greatest element is the most precise and the merge operation is meet.

$$\sigma^i[x] \quad = \quad \begin{cases} \{x\} & i = 0 \\ \{y \mid (*^i x, y) \in \sigma\} & i > 0 \end{cases}$$

(a) Lookup.

$$\begin{aligned} [x \overset{i}{\mapsto} y]\sigma &= \{(*^n w, z) \in t^*(\sigma') \mid n < k\} \\ &\text{where} \\ \sigma' &= \sigma \cup \{(*^i x, y)\} \cup \{(*^{i+j} x, z) \mid (*^j y, z) \in \sigma\} \\ t(\sigma) &= \{(*^{m+n} w, z) \mid \{(*^m w, x), (*^n x, z)\} \subseteq \sigma'\} \end{aligned}$$

(b) Insert. Adding an alias relation triggers the addition of any appropriate transitive relations. The constant k bounds the size of the store. The expression t^* computes the transitive closure of t.

$$\begin{aligned} \sigma \setminus [x \overset{i}{\mapsto} y] &= \sigma' \setminus \{(*^n w, z) \mid w \overset{n}{\rightsquigarrow} z \notin \sigma'\} \\ &\text{where} \\ \sigma' &= \sigma \setminus \{(*^i x, y)\} \\ x_0 \overset{n}{\rightsquigarrow} x_{n-1} \in \sigma &\Rightarrow \left(\bigcup_{i=0}^{n-2} \{(*^1 x_i, x_{i+1})\} \right) \subseteq \sigma \end{aligned}$$

(c) Delete. Removing an alias relation triggers the removal of any appropriate transitive relations.

Figure 4.3: Store operations. Each operation is implicitly lifted to operate on sets of variables.

This property would cause the concrete execution to diverge. We therefore restrict the store so that all the alias relations' *dereference exponents i* are less than some constant k (i.e., $\forall(*^i x, y) \in \sigma : i < k$). The constant k must be greater than the maximum number of dereferences that syntactically appear in any of the program's statements, to ensure that the store remains closed for all the operations the program might perform.

The delete operation $\sigma \setminus [x \overset{i}{\mapsto} y]$ removes the specified relation from the store. This operation also removes any transitive relations invalidated by removal.

The semantics make use of the lookup, insert, and delete operations. We also extend these operations to operate on sets of variables.

Configurations. A *configuration* pairs a program point with a store:

$$\tau \in \mathit{Configuration} : \mathit{ProgPoint} \times \mathit{Store}$$

A configuration represents a single node in a program trace. Individual components of a configuration are referred to using field-access notation: $\langle \rho_i, \sigma_i \rangle.\rho = \rho_i$ and $\langle \rho_i, \sigma_i \rangle.\sigma = \sigma_i$. Configurations are ordered point-wise:

$$\langle \rho, \sigma \rangle \sqsubseteq \langle \rho', \sigma' \rangle \Leftrightarrow (\rho = \rho') \wedge (\sigma \sqsubseteq \sigma')$$

Traces. A trace is an ordered sequence of configurations that represents a history of program execution:

$$T \in \mathit{Trace} : \overrightarrow{\mathit{Configuration}}$$

The empty trace is denoted by ε. The concatenation operation $T{:}\tau$ appends a configuration to a trace. We write $T \preceq_{\mathbb{P}} T'$ if T is a prefix of T'. Traces are ordered as follows:

$$\tau_1 {:} \cdots {:} \tau_n \sqsubseteq \tau'_1 {:} \cdots {:} \tau'_n \Leftrightarrow \tau_i \sqsubseteq \tau'_i$$

Note that two traces are comparable only if they contain the same sequence of program points.

The set of all possible traces induced by a given program defines the program's computation tree. We delay a formal definition of computations until we have formally defined the program semantics.

4.4.4 Semantics

The semantics is a nondeterministic big-step operational semantics [73]. The transition relation BASE on configurations is defined in Figure 4.4. The program *Pr* is globally defined.

A **skip** statement (Rule SKIP) creates a new configuration for each program point ρ' in the goto set $\mathcal{P}$ of the current program point ρ. The unmodified store is copied to each new program point.

An assignment statement (Rule ASSIGN) updates the store to map the variable represented by the left-hand side of the assignment to the value represented by the right-hand side. The value represented by the right-hand side is computed using the function $[\![e]\!]\sigma$, defined just below Rule ASSIGN.

Rule ASSIGN contains added complexity, because the base semantics apply to both concrete and abstract executions. In a concrete execution, the *l*-value of an assignment corresponds to only one variable. Every concrete assignment overwrites the old value (indicated in Rule ASSIGN as *kill*) with the new value. This operation is called *strong update*.

In an abstract execution, the *l*-value may correspond to multiple variables. In this case, the semantics must conservatively merge the old and new values. This operation is called *weak update*. Because weak update performs a merge, it loses

$$\frac{Pr(\rho) = \langle \textbf{skip}, \mathcal{P} \rangle \qquad \rho' \in \mathcal{P}}{\langle \rho, \sigma \rangle \xrightarrow{\text{BASE}} \langle \rho', \sigma \rangle} \tag{SKIP}$$

$$\frac{\begin{array}{c} Pr(\rho) = \langle *^n x \textbf{:=} e, \mathcal{P} \rangle \qquad \rho' \in \mathcal{P} \\ kill = \begin{cases} [\sigma^n[x] \xrightarrow{1} \sigma^{n+1}[x]] & : |\sigma^n[x]| = 1 \\ \emptyset & : otherwise \end{cases} \\ \sigma' = [\sigma^n[x] \xrightarrow{1} [\![e]\!]\sigma](\sigma \setminus kill) \end{array}}{\langle \rho, \sigma \rangle \xrightarrow{\text{BASE}} \langle \rho', \sigma' \rangle} \tag{ASSIGN}$$

$$\begin{array}{rcl} [\![\&x]\!]\sigma & = & \{x\} \\ [\![*^n x]\!]\sigma & = & \sigma^{n+1}[x] \end{array}$$

Figure 4.4: Intraprocedural reference model semantics.

precision. The abstract semantics can avoid this loss of precision when the statement's *l*-value corresponds to only one variable.

Computation Trees. We model a computation tree as the set of traces generated by executing a program's statements. Each computation step generates a new tree where each new trace is extended by one application of BASE:

$$\frac{\Gamma' = \{T{:}\tau{:}\tau' \mid T{:}\tau \in \Gamma \wedge \tau \xrightarrow{\text{BASE}} \tau'\}}{\Gamma \xrightarrow{C} \Gamma \cup \Gamma'} \tag{C}$$

Given a set of traces Γ, executing a statement extends each trace in Γ with a new configuration τ'. The new set of traces Γ' is unioned with the previous set Γ to create a new computation tree. Thus the tree contains a history of all the traces generated by each step of a program's execution. Each trace has finite length, but a tree may contain an infinite number of traces. The complete tree is computed by taking the transitive closure of C rooted at the initial trace set $\{\langle \rho_0, \emptyset \rangle\}$. Figure 4.5

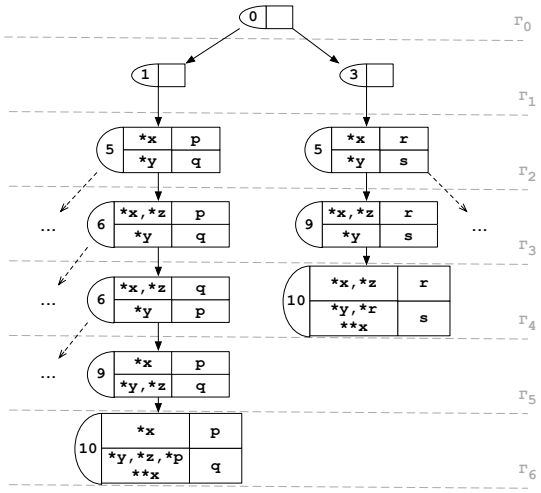


Figure 4.5: A partial computation tree for the program in Figure 4.2c. Each dotted horizontal line represents one computation step and each node represents one configuration. The stores are organized as a two-column table; for each row, the expressions in the first column are aliased with the expressions in the second column.

contains a partial computation tree for the program in Figure 4.2. Note how the non-deterministic semantics generate different paths to program point 10.

Transformations. A transformation t of a computation C forms a new transition relation $t(C)$. Transformation t is defined by a function f_t that modifies a set of traces to compute the tree for the next step of the computation. These modified traces introduce abstractions based on the current or previous computation steps. Our framework transforms the semantics by modifying the set of traces generated at each computation step. Such transformations are used by the framework to specify over-approximations of the concrete semantics.

Given a transformation function f_t, a new transition relation for computations is defined as follows:

$$\frac{\Gamma \xrightarrow{C} \Gamma'}{\Gamma \xrightarrow{t(C)} f_t(\Gamma')} \qquad (t)$$

The new transition relation uses f_t to transform the concrete tree Γ' into an abstract tree.

Section 4.5 describes three transformation functions that generate safe over-approximations of their inputs. These transformations can be composed to describe a broad spectrum of pointer analysis approximations.

4.5 Intraprocedural Pointer Analysis

The purpose of pointer analysis is to answer queries about possible aliasing between pointer expressions. In the intraprocedural case, we consider pointer analyses that determine—at a specific program point—whether the store contains a specific alias relation. Determining whether the semantics generates a store that contains a given alias relation corresponds to a query over the computation tree [20, 73]. More formally, given a query $\phi = \langle \rho, (*^i x, y) \rangle$ the analysis determines if the computation tree contains a trace whose final store contains the alias relation:

$$\Gamma \models \langle \rho, (*^i x, y) \rangle \;\; \Leftrightarrow \;\; \exists\, T{:}\langle \rho, \sigma \rangle \in \Gamma \;\mid\; y \in \sigma^i[x]$$

The result of a query on the computation tree generated by the base semantics is equivalent to an FS-MOP analysis for that query.

Because a precise solution is intractable, analyses must approximate the FS-MOP solution. Section 4.2 described several over-approximations of FS-MOP. These approximations are usually described algorithmically. In this section, we analyze the approximations to identify a small set of primitive semantic transformations that can be combined to specify the existing approximations. In addition,

$$\frac{\begin{array}{c} T{:}\tau \in \Gamma \\ \Gamma' = \{T' \in \Gamma \mid T' \overset{T}{\equiv} T{:}\tau\} \\ \sigma' = \bigsqcup\{\tau'.\sigma \mid T'{:}\tau' \in \Gamma'\} \end{array}}{T{:}\langle \tau.\rho, \sigma' \rangle \in f_{\text{ITE}}(\Gamma)} \text{ (ITE)}$$

$$\frac{\begin{array}{c} T{:}\tau_1{:}\tau \in \Gamma \\ \rho' \in \{\tau.\rho\} \cup \mathcal{F}[(\tau_1.\rho)] \end{array}}{T{:}\tau_1{:}\langle \rho', \tau.\sigma \rangle \in f_{\text{ICF}}(\Gamma)} \text{ (ICF)}$$

$$\frac{\begin{array}{c} T{:}\tau \in \Gamma \\ \sigma' = \bigsqcup_{x \in dom(\tau.\sigma)} I\mathcal{VE}(x, \tau) \end{array}}{T{:}\langle \tau.\rho, \sigma' \rangle \in f_{\text{IVE}}(\Gamma)} \text{ (IVE)}$$

$$\begin{array}{rcl} I\mathcal{VE}(x, \langle \rho, \sigma \rangle) & = & \{(*^i x', y') \mid x' \in \text{PE} \wedge y' \in \text{LE}_i\}, \text{ where} \\ \text{PE} & = & \{x' \mid x \overset{\text{PE}}{\equiv}_\sigma x'\} \\ \text{LE}_i & = & \{y' \mid y \in \sigma^i[PE] \wedge y \overset{\text{LE}}{\equiv}_\sigma y'\} \end{array}$$

Figure 4.6: Intraprocedural trace transformations.

we generalize existing approaches by converting binary design choices into continuous spectra of options, where possible. The transformations are designed so that any approximation defined using the primitives is sound by construction.

4.5.1 FS-MOP vs FS-MFP

In this section we explain how the FS-MOP and FS-MFP approximations are in fact endpoints of a continuum of approximations and introduce *Induced Trace Equivalence* as a technique for specifying arbitrary points along this continuum.

FS-MOP computes a separate result for each path through the CFG. There are potentially an infinite number of paths, and even when the number of paths is finite it is still exponential in the number of branches in the CFG. An FS-MFP pointer analysis mitigates this problem by merging results from separate paths when they reach a common program point. In essence, the FS-MFP analysis groups paths into a polynomial number of equivalence classes.

We abstract and generalize this approximation mechanism into a transformation on the computation tree called Induced Trace Equivalence (ITE). This transformation partitions individual computation paths (i.e., traces) into arbitrary equivalence classes based on a given equivalence relation for traces. ITE is a general semantic transformation that accumulates pointer information by unioning stores from equivalent traces. In the transformed semantics, a new configuration is the union of the final store of all equivalent traces in the underlying semantics. The equivalence relation must respect the ordering on traces:

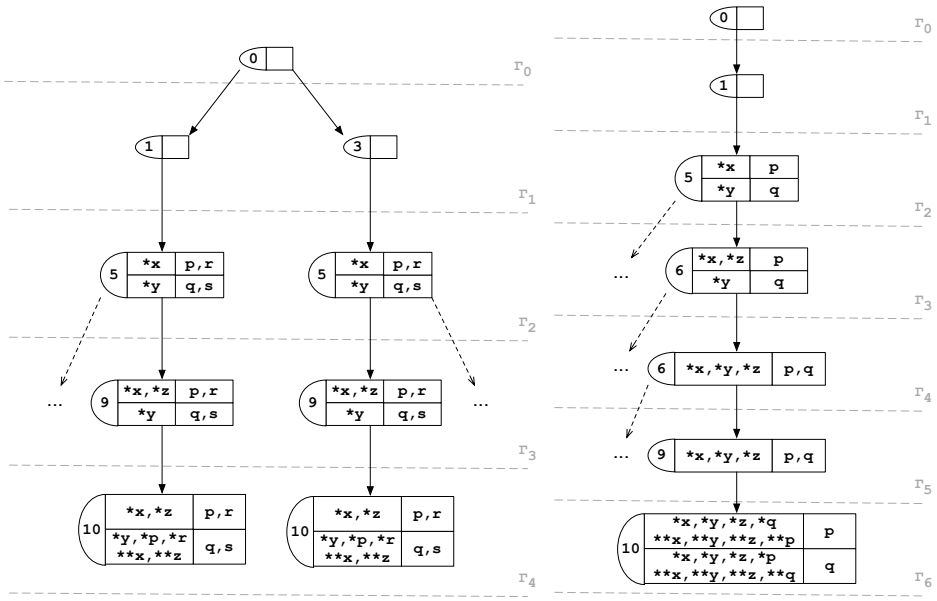
$$\frac{T_1 \overset{T}{\equiv} T_2 \qquad T_1 \sqsubseteq T_1' \qquad T_2 \sqsubseteq T_2'}{T_1' \overset{T}{\equiv} T_2'}$$

The Induced Trace Equivalence transformation is formally defined by Rule ITE in Figure 4.6. For each trace generated by the underlying semantics, ITE replaces the last store in the trace with the union of all final stores from equivalent traces.

There are several existing algorithms for FS-MFP [44, 80] that differ greatly in their implementation. However, the precision of their results are equivalent and the following ITE equivalence relation captures their equivalence:

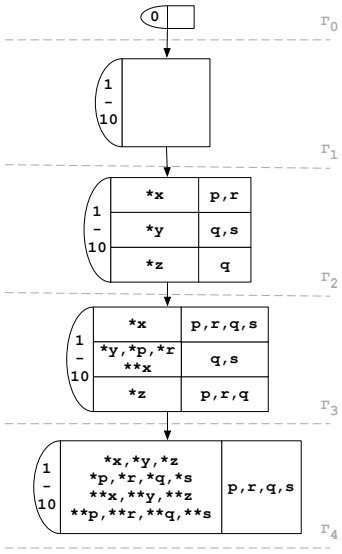
$$T_1{:}\langle \rho, \sigma_1 \rangle \overset{T}{\equiv} T_2{:}\langle \rho, \sigma_2 \rangle$$

The effect of this transformation on the reference model is shown in Figure 4.7a.



(a) FS-MFP using ITE.

(b) Memoization using ITE.



(c) FI-MFP using ITE and ICF.

Figure 4.7: Transformed program trees that compute approximations using ITE (for FS-MFP and memoization) and ITE composed with ICF (for FI-MFP). The graphs use the same conventions as Figure 4.5.

ITE is a powerful abstraction that can specify an arbitrary point in the continuum that lies between FS-MOP and FS-MFP simply by changing the equivalence relation to modulate those computation paths that are considered equivalent. A practical example of one point in this space is *alias instances*, as described by Hind et al. [44]. Many other approximations can be created by parameterizing ITE with different equivalence relations. For example, loops can be unrolled or even partitioned arbitrarily (e.g., partitioned into even and odd iterations). In addition to looking at program points, the equivalence relation can also use stores to determine equivalence.

ITE is general enough to capture other abstractions, such as *memoization*. Memoization is a classic widening operator that forces the computation to accumulate pointer information at a program point that appears multiple times in a single path (i.e., program points within a loop). In Figure 4.7b, when program point 6 is encountered for the second time, it accumulates the information from the previous visit to program point 6. ITE can specify memoization by using the equivalence relation:

$$\frac{T_1 \preceq_{\mathbb{P}} T_2 \vee T_2 \preceq_{\mathbb{P}} T_1 \qquad T_1 = T_1' : \langle \rho, \sigma_1 \rangle \qquad T_2 = T_2' : \langle \rho, \sigma_2 \rangle}{T_1 \overset{T}{\equiv} T_2}$$

ITE can also be used to specify a points-to-based pointer analysis, rather than an alias-based pointer analysis. Recall from Section 4.2 that an alias-based FS-MFP pointer analysis with strong-update semantics is strictly more precise than its points-to-based equivalent. The reason for this difference is that an alias-based analysis may, in some instances, maintain distinct results for distinct paths; whereas a points-to-based analysis implicitly merges results from distinct paths. For example, consider a join point that is reached along one path with results $(*x, y)$ and along another path with results $(*y, z)$. The points-to based analysis does not explicitly

maintain information about transitive aliases, so it must answer queries by taking the transitive closure of the points-to relation — since at the join point there exists a points-to relation from x to y and another from y to z, a points-to-based analysis must (imprecisely) conclude that $*^2x$ aliases z. By contrast, the alias-based analysis can be more precise, because it explicitly maintains information about transitive aliases — while the alias relations $(*x, y)$ and $(*y, z)$ both exist at the join point, the relation $(*^2x, z)$ does not, and the analysis correctly concludes that $*^2x$ does not alias z.

Since the aliased-based analysis is more precise than the points-to-based one, we can approximate points-to with a transformation. We modify ITE to take as a parameter a join operation, which is used to join the final stores of equivalent paths. Alias analysis uses set union as the join operator. Points-to analysis uses a join operator that joins two stores, removes any relations of the form $(*^n x, y)$ where $n > 1$, then computes the transitive closure of the result.

4.5.2 Flow-Sensitivity vs Flow-Insensitivity

A flow-sensitive analysis respects a program's control flow as embodied by the CFG. In contrast, a flow-insensitive analysis ignores a program's control flow and assumes that any statement can be executed after any other statement. This approximation can be expressed by adding additional edges to the CFG to make it a complete graph, including reflexive edges. We abstract and generalize this approximation mechanism into a transformation on the computation tree called *Induced Control Flow* (ICF). This transformation allows us to add additional control-flow beyond that specified by the control-flow graph.

Rule ICF in Figure 4.6 formalizes this behavior by defining transformation function f_{ICF}. The transformation relies on an ICF map $\mathcal{F} : \mathit{ProgPoint} \rightarrow \overline{\mathit{ProgPoint}}$,

which defines the degree of control-flow abstraction. If $Pr(\rho) = \langle stmt, \mathcal{P} \rangle$ and $\mathcal{F}(\rho) = \mathcal{P}'$, then execution proceeds non-deterministically from ρ to any program point in $\mathcal{P} \cup \mathcal{P}'$.

ICF can specify an FI-MOP analysis using the following map:

$$\forall \rho \in \mathit{ProgPoint} : \mathcal{F}[\rho] = \mathit{ProgPoint}$$

Since $\mathcal{F}$ maps each program point to every existing program point, queries on the transformed computation tree are equivalent to FI-MOP. By changing $\mathcal{F}$ we can use ICF to vary the precision of the pointer analysis along a spectrum between FS-MOP and FI-MOP.

The two transformations ITE and ICF can be used in combination to specify even more levels of precision, for example FI-MFP. There are many existing algorithms for FI-MFP [29, 37] that have very different implementations. However, their results are equivalent and those results can be specified by using the ITE equivalence relation MFP and the ICF map $\mathcal{F}$ as given above. Figure 4.7c gives an example. Note that at each computation step of the FI-MFP analysis, every program point has the same store.

By selectively adding edges to the original CFG using ICF and varying the ITE equivalence relation, we can tune the analysis at a very fine granularity of precision anywhere between FS-MOP and FI-MFP. In addition, by extending ITE to have a separate equivalence relation *per variable* we could control the precision on a per-variable basis. This change would enable the framework to specify analyses such as Guyer et al.'s client-driven pointer analysis [34], which uses per-variable flow-sensitivity.

4.5.3 Variable Equivalence

This section shows how a set of existing pointer analyses, which use different abstractions of pointer information, can be unified by a single transformation, which we call *Induced Variable Equivalence*. In particular, several analyses — including Steensgaard's near linear-time analysis [78], Shapiro and Horwitz's family of analyses [75], and Das' One-Level Flow analysis [25] — assume an FI-MFP analysis as a starting point but differ in how they abstract pointer information.

We distinguish two kinds of pointer information abstractions: *pointer equivalence* and *location equivalence*. Variables x and y are pointer equivalent if they point to the same values. Variables x and y are location equivalent if they are pointed to by the same variables.

Steensgaard's analysis maintains an invariant on pointer relations requiring that no pointer equivalence class can point to more than one location equivalence class. The Shapiro-Horwitz family of analyses can be specified as a simple extension to Steensgaard's analysis: randomly assign k labels to the program variables and only apply Steensgaard's invariant to memory locations with the same label, thereby enforcing the property that no pointer equivalence class can point to more than k location equivalence classes (when $|k|$ is equal to the number of variables, this analysis is equivalent to FI-MFP). Das' One-Level Flow analysis treats top-level pointers (those that have nothing pointing to them) similarly to FI-MFP but all other pointers similarly to Steensgaard.

We generalize these restrictions on pointer- and location-equivalence classes as a transformation called Induced Variable Equivalence (IVE). This transformation forces variables to be pointer and/or location equivalent according to the equivalence relations $\overset{\text{PE}}{\equiv}_\sigma$ and $\overset{\text{LE}}{\equiv}_\sigma$ given to the analysis. Given a store, variables x and y are

pointer equivalent with respect to σ (denoted $x \overset{\text{PE}}{\equiv}_\sigma y$) if $\sigma^1[x] = \sigma^1[y]$. Variables x and y are location equivalent (denoted $x \overset{\text{LE}}{\equiv}_\sigma y$) if $\forall z \in dom(\sigma) : x \in \sigma^1[z] \Leftrightarrow y \in \sigma^1[z]$.

We formally define the transformation with Rule IVE in Figure 4.6, which updates the store of a generated configuration to obey given pointer and location equivalence relations. For convenience we define the equivalence relation $\overset{\text{LPE}}{\equiv}_\sigma$ as $x \overset{\text{LPE}}{\equiv}_\sigma y \Leftrightarrow x \overset{\text{PE}}{\equiv}_\sigma y \wedge x \overset{\text{LE}}{\equiv}_\sigma y$. The combined relation must respect the ordering on stores:

$$\frac{x \overset{\text{LPE}}{\equiv}_\sigma x' \qquad \sigma \sqsubseteq \sigma'}{x \overset{\text{LPE}}{\equiv}_{\sigma'} x'}$$

For simplicity, our framework has assumed a field-insensitive pointer semantics. Our framework can vary field-sensitivity using IVE by dividing struct fields into different partitions and making all fields in the same partition both pointer and location equivalent. A practical example of this capability is the *normalize*, *lookup*, and *resolve* functions defined by Yong et al. [83].

By enforcing the following invariant on the IVE equivalence relations we correctly specify Steensgaard's analysis:

$$\frac{p \in dom(\sigma) \qquad \{q,r\} \subseteq \sigma^1[p]}{q \overset{\text{LPE}}{\equiv}_\sigma r}$$

The invariant states that if a pointer points to two different memory locations, those memory locations are both pointer- and location-equivalent.

We can specify the One-Level Flow analysis using the following two invariants:

$$\frac{\begin{array}{l} Pr(\rho_1) = (p := \&q, \mathcal{P}_1) w \in \sigma^1[x] \\ Pr(\rho_2) = (p := \&r, \mathcal{P}_2) y, z\} \subseteq \sigma^1[w] \end{array}}{q \overset{\text{LPE}}{\equiv}_\sigma r \qquad y \overset{\text{LPE}}{\equiv}_\sigma z}$$

Currently, no approximations exploit the ability to separate the notions of pointer equivalence and location equivalence. An interesting area for future work is to create new approximations by treating these two notions independently.

4.6 Interprocedural Reference Model

We extend the reference model introduced in Section 4.4 to include multiple procedures with local variables. For clarity the language is kept simple but could be extended to include more features as needed.

4.6.1 Overview

The extended language of this section permits multiple procedures and provides call and return statements. In a language with procedures, variable names are not equivalent to variable locations, because each procedure invocation must be distinct. Each invocation of a procedure generates new locations (i.e., *addresses*) for the procedure's formal parameters and local variables. A variable's value is determined by first retrieving the variable's address for the current invocation, then looking up the value of that address in the store. Thus the interprocedural semantics require an extra level of indirection in the memory model.

Because each procedure invocation maps its variables to addresses, the interprocedural semantics also require a *stack*. The stack maintains variable addresses for nested procedure calls. A procedure call creates new addresses for each of the procedure's variables and pushes this information on the stack. A procedure return pops this information off the stack.

The remainder of this section gives the formal definitions for the syntax, program domain, and semantics of the interprocedural reference model. These def-

$$
\begin{array}{rcl}
 & n \in \mathbb{N} \quad x \in \mathit{Variable} \quad \rho \in \mathit{ProgPoint} & \\
\mathsf{pe} \in \mathit{ProcEntry} & \subset & \mathit{ProgPoint} \\
e \in \mathit{PtrExpr} & ::= & \&x \mid *^n x \\
s \in \mathit{Stmt} & ::= & *^n x := e \mid \textbf{skip} \mid \\
 & & *^n x := \textbf{call}\ \mathsf{pe}(e) \mid \textbf{return}\ e \\
\mathit{Pr} \in \mathit{Program} & : & \mathit{ProgPoint} \to \mathit{Stmt} \times \overline{\mathit{ProgPoint}}
\end{array}
$$

Figure 4.8: An unstructured pointer language with procedures.

initions are similar to those of Section 4.4, but have been extended to incorporate the level of indirection required to accommodate addresses.

4.6.2 Syntax

Figure 4.8 extends the reference model language to include procedure calls and returns. A subset of the program points are distinguished as procedure entry points; each procedure has a unique entry point, and there is exactly one procedure entry point that is also the entire program's entry point. Each procedure also has a unique exit point, mapped to either $\langle \textbf{skip}, \emptyset \rangle$ or $\langle \textbf{return}\ e, \emptyset \rangle$.

A procedure body is the set of program points reachable from a procedure entry point without transferring control through a **call** or **return**. Each procedure has a formal parameter and a set of local variables associated with it. We define the following mappings:

$\mathit{body} : \mathit{ProgPoint} \to \overline{\mathit{ProgPoint}}$ Given a program point ρ, yields all program points in the body of the procedure that contains ρ.

$\mathit{param} : \mathit{ProcEntry} \to \mathit{Variable}$ maps a procedure entry point to the procedure's formal parameter.

$locals : ProcEntry \rightarrow \overline{Variable}$ maps a procedure entry point to the variables that syntactically appear in the procedure body.

The semantics sometimes must refer to all variables that a procedure directly references. For convenience, we define this set as $vars(\mathsf{pe}) = \{param(\mathsf{pe})\} \cup locals(\mathsf{pe})$. We assume this set is distinct for each procedure.

4.6.3 Semantic Domain

The semantic domain is similar to that of the intraprocedural model, except the memory model includes addresses.

Addresses. An address provides a location for a variable. Each formal and local variable of a procedure is mapped to a new address for each distinct procedure invocation. The domain *Address* is potentially infinite and totally ordered.

Address Maps. An *address map* provides addresses for variables:

$$m \in AddressMap : Variable \rightarrow Address$$

Each invocation of a procedure generates an address map for the variables directly referenced by that procedure. An address map m implicitly defines an inverse map m^{-1} which provides variable names for addresses. We define the function $rng(m)$ to mean all the addresses in map m.

Stores. Stores now describe relationships among addresses:

$$\sigma \in Store : \overline{\mathbb{N} \times Address \times Address}$$

Stores and address maps work together to provide a variable's value. Given an address map m and a store σ, the addresses that alias $*^i x$ are given by $\sigma^i[m[x]]$. We lift store lookup and update to operate on sets of addresses.

$$\frac{Pr(\rho) = \langle \textbf{skip}, \mathcal{P} \rangle \qquad \rho' \in \mathcal{P}'}{\langle \rho, \sigma, A \rangle \xrightarrow{\text{P-BASE}} \langle \rho', \sigma, A \rangle} \quad \text{(P-SKIP)}$$

$$\frac{\begin{array}{c} Pr(\rho) = \langle *^n x \textbf{:=} e, \mathcal{P} \rangle \qquad \rho' \in \mathcal{P}' \\ kill = \begin{cases} [\sigma^n[\alpha[x]] \xmapsto{1} \sigma^{n+1}[\alpha[x]]] & : |\sigma^n[\alpha[x]]| = 1 \\ \emptyset & : otherwise \end{cases} \\ \sigma' = [\sigma^n[\alpha[x]] \xmapsto{1} [\![e]\!]\sigma\alpha](\sigma \setminus kill) \end{array}}{\langle \rho, \sigma, \alpha{:}A \rangle \xrightarrow{\text{P-BASE}} \langle \rho', \sigma', \alpha{:}A \rangle} \quad \text{(P-ASSIGN)}$$

$$\begin{array}{rcl} [\![\&x]\!]\sigma\alpha & = & \{\alpha[x]\} \\ [\![*^n x]\!]\sigma\alpha & = & \sigma^{n+1}[\alpha[x]] \end{array}$$

$$\frac{\begin{array}{c} Pr(\rho) = \langle *^n x \textbf{:=call } \mathsf{pe}(e), \mathcal{P} \rangle \\ \alpha' = (\rho, \bigsqcup_{x \in vars(\mathsf{pe})} [x \xmapsto{1} fresh]) \\ \sigma' = [\alpha'[param(\mathsf{pe})] \xmapsto{1} [\![e]\!]\sigma\alpha]\sigma \end{array}}{\langle \rho, \sigma, \alpha{:}A \rangle \xrightarrow{\text{P-BASE}} \langle \mathsf{pe}, \sigma', \alpha'{:}\alpha{:}A \rangle} \quad \text{(P-CALL)}$$

$$\frac{\begin{array}{c} Pr(\rho) = \langle \textbf{return } e, \emptyset \rangle \\ Pr(\alpha.\rho) = (*^n x \textbf{:=call } \mathsf{pe}(e_c), \mathcal{P}') \qquad \rho' \in \mathcal{P}' \\ \sigma' = [\sigma^n[\alpha'[x]] \xmapsto{1} [\![e]\!]\sigma\alpha]\sigma \end{array}}{\langle \rho, \sigma, \alpha{:}\alpha'{:}A \rangle \xrightarrow{\text{P-BASE}} \langle \rho', \sigma', \alpha'{:}A \rangle} \quad \text{(P-RET)}$$

Figure 4.9: Interprocedural reference model semantics.

The following order relation on store, address-map pairs helps prove soundness:

$$\begin{aligned}
&\langle \sigma_1, m_1 \rangle \sqsubseteq \langle \sigma_2, m_2 \rangle \Leftrightarrow \\
&\quad \forall x \in dom(m_1) : (*^i m_1[x], a_2) \in \sigma_1 \Rightarrow \\
&\quad\quad \exists (*^i m_2[x], a_2') \in \sigma_2 \mid m_2^{-1}[a_2'] = m_1^{-1}[a_2]
\end{aligned}$$

Frames. Each procedure invocation is associated with a *frame*. A frame is a program-point, address-map pair:

$$\alpha \in Frame \quad : \quad ProgPoint \times AddressMap$$

A frame provides context for a procedure invocation. The program-point element corresponds to the statement that invoked the current procedure. The address-map element provides addresses for each variable directly referenced by the current procedure. A frame's elements are referenced with field access notation. For convenience we write $\alpha[x]$ to mean $\alpha.m[x]$ and $\alpha^{-1}[x]$ to mean $\alpha.m^{-1}[x]$.

Stacks. A stack is an ordered sequence of frames:

$$A \in Stack : \overrightarrow{Frame}$$

A procedure call pushes a new frame on the stack; returning from a procedure pops that procedure's frame from the stack.

Configurations. The configuration structure and order relation are extended from the intraprocedural model to include a stack:

$$\tau \in Configuration : ProgPoint \times Store \times Stack$$

$$\langle \rho, \sigma, \langle \rho_m, m \rangle : A \rangle \sqsubseteq \langle \rho', \sigma', \langle \rho'_m, m' \rangle : A' \rangle \Leftrightarrow$$
$$(\rho = \rho') \wedge (\langle \sigma, m \rangle \sqsubseteq \langle \sigma', m' \rangle) \wedge (\rho_m = \rho'_m)$$

4.6.4 Semantics

Figure 4.9 defines the configuration transition relation P-BASE for **skip**, assignment, **call**, and **return**. Rules P-SKIP and P-ASSIGN propagate the stack without modifying it. The expression $\alpha{:}A$ in the conclusion of rule P-ASSIGN refers to a stack whose top element is α and whose remaining elements are A. Variable assignment has the same strong/weak update semantics as in Section 4.4.4, extended to incorporate an address map. The expression evaluation function $[\![]\!]$ operates over both a store and an address map.

Rule P-CALL applies to **call** statements. The semantics creates a new frame for the procedure invocation by generating fresh addresses for the invoked procedure's formal and local variables. The argument values are bound to the formal parameter values. The new frame is pushed on the stack, and control is transferred to the invoked procedure's entry point.

Rule P-RET applies to **return** statements. The callee's return expression is evaluated in its frame, and the results are stored in the caller's frame. The callee frame is popped off the stack, and control is transferred to the program points in the calling statement's goto set.

4.7 Interprocedural Pointer Analysis

Interprocedural pointer analysis is a variation of the pointer analysis of Section 4.5 modified to operate on configurations that contain address maps. Interprocedural FS-MOP pointer analysis may not terminate, due to recursive calls, heap

allocation, or loops that contain callsites. The analysis designer must ensure that an approximation guarantees termination. Section 4.8.2 discusses the necessary termination conditions and suggests how to ensure them.

Figure 4.10 gives the formal definitions of the interprocedural transformation functions. With two exceptions, the interprocedural transformations are identical to the intraprocedural ones, modulo address map operations. We provide an informal overview of the these differences, then show how these small modifications permit a wider range of pointer analysis approximations.

Induced Trace Equivalence. Interprocedural ITE extends intraprocedural ITE to merge address maps as well as stores. The variables of distinct, concrete invocations are distinct. Interprocedural ITE can blur this distinction, in order to tune context sensitivity. Section 4.7.1 describes several such approximations.

Rule P-ITE in Figure 4.10 defines a function $\mathcal{M}$, which merges address maps in a way that permits ITE to ensure termination conditions. The function forces variables for equivalent contexts to have equivalent addresses in the store. We omit the technical details for brevity; they can be found in [36].

Induced Control Flow. Interprocedural ICF is constrained so that it adds additional control flow only *within* the body of a procedure, never *between* procedures. However, control flow can still pass between arbitrary statements in the program simply by following the correct sequence of calls and returns between the two statements.

Induced Variable Equivalence. Interprocedural IVE extends intraprocedural IVE to operate on address maps. The IVE equivalence relations $\overset{\text{PE}}{\equiv}_\sigma$ and $\overset{\text{LE}}{\equiv}_\sigma$ are on variables, but stores now contain addresses. Function $\mathcal{IVE}$ translates between these

$$\frac{\begin{array}{c} T{:}\tau \in \Gamma \qquad \tau = \langle \rho, \sigma, \langle \rho_a, m_a \rangle {:} A \rangle \\ \Gamma' = \{T' \in \Gamma \mid T' \overset{T}{\equiv} T{:}\tau\} \\ \sigma' = \bigsqcup \{\tau'.\sigma \mid T{:}\tau' \in \Gamma'\} \\ f_m = \mathcal{M}(\{\alpha'.m \mid T{:}\tau' \in \Gamma' \wedge \tau'.A = \alpha'{:}A'\}) \end{array}}{T{:}\langle \tau.\rho, f_m(\sigma'), \langle \rho_a, f_m \circ m_a \rangle {:} A \rangle \in f_{\text{ITE}}(\Gamma)} \quad \text{(P-ITE)}$$

$$\frac{\begin{array}{c} T{:}\tau_1{:}\tau \in \Gamma \\ \rho' \in \{\tau.\rho\} \cup \mathcal{F}[(\tau_1.\rho)] \\ \mathcal{F}[\tau_1.\rho] \subseteq \mathit{body}(\tau_1.\rho) \end{array}}{T{:}\tau_1{:}\langle \rho', \tau.\sigma, \tau.A \rangle \in f_{\text{ICF}}(\Gamma)} \quad \text{(P-ICF)}$$

$$\frac{\begin{array}{c} T{:}\tau \in \Gamma \qquad \tau = \langle \rho, \sigma, \alpha{:}A \rangle \\ \sigma' = \bigsqcup_{x \in \mathit{dom}(\alpha.m)} I\mathcal{VE}(x, \tau) \end{array}}{T{:}\langle \tau.\rho, \sigma', \tau.A \rangle \in f_{\text{IVE}}(\Gamma)} \quad \text{(P-IVE)}$$

$$\begin{array}{rcl} I\mathcal{VE}(x, \langle \rho, \sigma, \alpha{:}A \rangle) & = & \{(*^i a, a') \mid a \in \text{PE} \wedge a' \in \text{LE}_i\}, \text{ where} \\ \text{PE} & = & \{\alpha[x'] \mid x \overset{\text{PE}}{\equiv}_{\sigma\alpha} x'\} \\ \text{LE}_i & = & \{\alpha[y'] \mid a_y \in \sigma^i[PE] \wedge \alpha^{-1}[a_y] \overset{\text{LE}}{\equiv}_{\sigma\alpha} y'\} \end{array}$$

$$\begin{array}{rcl} f_m(\sigma) & = & \{(*^i f_m(a_1), f_m(a_2)) \mid (*^i a_1, a_2) \in \sigma\} \\ \mathcal{M}(m^*) & = & \bigcup_{m \in m^*} \bigcup_{x \in \mathit{dom}(m)} \begin{cases} (m[x], m_0[x]) & x \in \mathit{dom}(m_0) \\ (m[x], m[x]) & \text{otherwise} \end{cases} \\ & & \text{where } \forall m_i \in m^* : \mathit{min}(\mathit{rng}(m_0)) \leq \mathit{min}(\mathit{rng}(m_i)) \end{array}$$

Figure 4.10: Interprocedural trace transformations.

two domains to ensure that the store properly obeys the equivalence relations.

4.7.1 Context-Sensitivity

Surprisingly, these simple extensions to the intraprocedural version of the three transformations are sufficient to specify the precision of a host of interprocedural approximations. We show how the transformations can specify both call-string and functional context-sensitivity, along with a number of variations of the same.

4.7.1.1 Call-string Equivalence

The current *call-stack* (i.e., the sequence of procedure calls without matching returns that led to the current program point) is contained in the current configuration. Traditional call-string context-sensitivity is specified by using the call-stack to distinguish contexts, similarly to Emami et al. [28]. We can use ITE to union the stores of all paths to a procedure's entry point that contain identical call-stacks:

$$\frac{A = \alpha_1{:}\cdots{:}\alpha_n \qquad A' = \alpha'_1{:}\cdots{:}\alpha'_n \qquad \alpha_i.\rho = \alpha'_i.\rho}{T{:}\langle \mathsf{pe}, \sigma, A\rangle \overset{T}{\equiv} T'{:}\langle \mathsf{pe}, \sigma', A'\rangle}$$

In the presence of recursion there are an infinite number of possible call-stacks. The usual response is to only consider the k most recent calls on the stack, yielding the traditional *k-limited* context-sensitivity. However, our framework exposes other opportunities: k-limited context-sensitivity is fairly arbitrary — we can be more precise, while still maintaining a finite call-stack, by limiting the number of *repeated elements* on the call-stack. For example, instead of k-limiting the call-stack, we could k-limit the unrolling of recursive procedures. Setting k to 0 yields the common practice of collapsing recursive cycles in the call-graph and treating them context-insensitively.

Our framework can control the level of context-sensitivity at a very fine granularity. Guyer et al.'s client-driven pointer analysis [34] allows *per-procedure* context-sensitivity; our framework can capture this approximation by specifying the dual (i.e., per-procedure context-insensitivity). To make a procedure with entry point pe context-insensitive, we use the equivalence relation:

$$T_1{:}\langle \mathsf{pe}, \sigma_1, A_1 \rangle \overset{T}{\equiv} T_2{:}\langle \mathsf{pe}, \sigma_2, A_2 \rangle$$

Our framework can go further and specify context-sensitivity at a *per-call* level, meaning that the contexts for a given procedure are partitioned using the call-stack, with contexts in the same partition being treated context-insensitively with respect to each other. Many other ITE equivalence relations based on the call-stack are possible.

4.7.1.2 Functional Equivalence

Our framework can also determine ITE equivalence based on the contents of the store at procedure entry, specifying something similar to functional context-sensitivity. It can require that memory stores for equivalent contexts be functionally identical, similar to Wilson and Lam's *partial transfer functions* [82]:

$$T_1{:}\langle \mathsf{pe}, \sigma, A_1 \rangle \overset{T}{\equiv} T_2{:}\langle \mathsf{pe}, \sigma, A_2 \rangle$$

Our framework can also be used to describe *Object-sensitivity* [57], which is a technique for object-oriented programs that uses equivalent contexts for all method calls on the same receiver object. In object-oriented languages, a pointer to the receiver object, called **this**, is passed as an implicit parameter to all methods; by using the receiver object in the store to determine trace equivalence, our framework can specify object-sensitivity:

$$\frac{\sigma_1^1[\alpha_1[\textbf{this}]] = \sigma_2^1[\alpha_2[\textbf{this}]]}{T_1{:}\langle \mathsf{pe}, \sigma_1, \alpha_1{:}A_1\rangle \overset{T}{\equiv} T_2{:}\langle \mathsf{pe}, \sigma_2, \alpha_2{:}A_2\rangle}$$

4.7.1.3 Limitations

There is a class of sound context-sensitive approximation, known as *bottom-up context-sensitivity* [64], that our framework cannot represent. Its defining characteristic is that (1) like a context-sensitive analysis, information passed to a procedure from one call-site cannot be returned by that procedure to a different call-site; (2) unlike a full context-sensitive analysis, all contexts for a given procedure are merged when analyzing that procedure. Thus, bottom-up context sensitivity represents a mid-way point between full context-sensitivity and complete context-insensitivity.

Bottom-up context-sensitivity is achieved by processing a program from the bottom up (hence the name), i.e., it starts at the leaves of the call-graph and creates summaries of each function, repeatedly propagating information upwards through the call-graph. Our reference model only allows forward propagation of information, so it is unable to replicate the effects of bottom-up context-sensitivity.

4.7.2 Heap Model

Until now we have not addressed the issue of dynamic memory. The interprocedural reference model can accommodate it by defining a procedure *malloc* with entry point pe_m such that $\mathit{locals}(\mathsf{pe}_m) = \{x\}$ and $\mathit{body}(\mathsf{pe}_m) = \{\textbf{return}\ x\}$. Each call to *malloc* returns a fresh address, which models the potentially infinite heap space.

The infinite number of addresses that can be returned by *malloc* implies that the pointer analysis may never converge, therefore we need to be able to approxi-

mate the heap using a finite number of abstract memory locations. The interprocedural transformations can be used to specify a variety of heap abstractions.

The purpose of abstracting the heap is to represent the heap using a finite number of addresses. Each concrete *malloc* call is distinct and yields a new address, so abstracting the heap requires partitioning these addresses into a finite number of equivalence classes. Since each equivalence class maps to multiple concrete addresses, the semantics should always use weak updates for any variable that holds the result of a *malloc* call.

To accommodate this requirement we modify the definition of *malloc* to: $\mathit{locals}(\mathsf{pe}_m) = \{x, y\}$ and $\mathit{body}(\mathsf{pe}_m) = \{x := \&x, x := \&y, \textbf{return}\ x\}$. We then use ITE to ensure that the return value of *malloc* must point to multiple memory locations (i.e., both x and y), and therefore these return values will always be subject to weak updates.

$$\frac{\{\rho_1, \rho_2\} \subseteq \mathit{body}(\mathsf{pe}_m)}{T_1{:}\langle \rho_1, \sigma_1, A_1 \rangle \overset{T}{\equiv} T_2{:}\langle \rho_2, \sigma_2, A_2 \rangle} \qquad \text{(ITE-MALLOC)}$$

We can now specify several common heap models. In a context-sensitive heap model, *malloc* should return a fresh address for each *malloc* call-site in each distinct context; however, multiple traversals of the same *malloc* call-site in the same context should return the same address. A context-sensitive analysis as outlined in Section 4.7.1 automatically yields a context-sensitive heap model.

A less precise heap model treats each static allocation site (i.e., each *malloc* call-site) as a single abstract memory location, regardless of the call's surrounding context. This model can be specified using a modified form of the context insensitivity transformation from Section 4.7.1.1, which equates traces that end in the same *malloc* call-site.

The least precise heap model represents the entire heap as a single abstract memory location by forcing *malloc* to return the same address for every call. This model can be specified using a modified form of the context insensitivity transformation from Section 4.7.1.1, which equates traces that end with pe_m, the entry point of *malloc*.

The heap model can also be parameterized by a constant k. A k-limited heap model treats each static allocation site as k distinct abstract memory locations. The first $k-1$ calls are modeled with distinct addresses that map to distinct concrete addresses and can be strongly updated; all subsequent calls are modeled with a single address that may map to multiple concrete addresses and must be weakly updated. This model can be specified using a modified form of ITE-MALLOC that equates traces that end in a *malloc* call-site only if the trace contains $k-1$ instances of that same call-site.

The frameworks of Sagiv et al. [71] and Deutsch [26] characterize more expressive heap approximations than our heap model, because their frameworks operate on a richer abstract domain. The two frameworks describe incomparable precision classes [71]. We believe that with some additional effort, our model could be enriched so that it were more compatible with either Sagiv et al.'s or Deutsch's. This effort would increase the range of heap approximations that we could specify.

4.8 Soundness and Termination

All transformations describe sound analyses. However, an analysis only terminates under certain conditions. In this section, we outline a soundness proof and describe the properties required to prove termination.

4.8.1 Soundness

The ITE, ICF, and IVE transformations each generate sound over-approximations of the FS-MOP solution: all queries returning true in the base analysis are true in the transformed analysis, but the inverse does not necessarily hold. In this section we sketch a proof that every analysis described by a combination of transformations computes a sound approximation of the concrete computation.

Theorem 6 (Soundness). Let $t = t_n \circ \cdots \circ t_1 \circ C$ be the composition of n transformations of a computation, where $t_i \in \{\text{ITE}, \text{ICF}, \text{IVE}\}$. For all programs with initial configurations $\tau_0 = \langle \rho_0, \emptyset \rangle$ and all queries $\phi = \langle \rho, (*^i x, y) \rangle$: $C^*(\{\tau_0\}) \models \phi \Rightarrow t^*(\{\tau_0\}) \models \phi$.

Proof Sketch. First note that for each transformation function f_t, $\Gamma \sqsubseteq f_t(\Gamma)$ and that P-BASE is monotone. For the soundness condition to hold, the existence of a satisfying trace T in C must imply the existence of an overapproximation $T' \in t^*$ such that $T \sqsubseteq T'$. This property can be proved by an induction over the length of computation. □

4.8.2 Termination

A concrete computation of the reference model does not terminate on programs that contain cyclic control-flow graphs, because trees have an infinite number of traces. However, if the domain of configurations is finite then the computation trees are regular. In this case, we can employ a *summarization* technique which terminates the computation when every trace contains a repeated node [73].

The domain of configurations is finite for the intraprocedural but infinite for the interprocedural model, because address maps are always unique. An ITE

transformation—such as *k*-limiting the stack—can be employed to ensure termination by ensuring that all computation trees in the approximate semantics are regular.

4.9 Chapter Summary

In this chapter, we create a formal framework for describing the space of possible pointer analysis approximations. This framework is useful because it aids systematic exploration of this space and allows researchers to precisely characterize the precision of the various algorithms they devise.

The heart of the framework is the idea of *induced equivalence*. We define a base semantics that describes the most precise, NP-hard MOP pointer analysis. We then define three semantic transformations that (1) collapse variables together, (2) collapse program paths together, and (3) add additional control-flow. These transformations combine to create a new, *approximate* semantics that reduces the precision of the MOP analysis. However, in return the transformations guarantee invariants about the approximate analysis that researchers can take advantage of to create efficient pointer analysis algorithms (the particular invariants guaranteed depend on how the transformations are used; for example, a flow-insensitive analysis guarantees that every program point has an identical solution, and therefore only a single solution needs to be computed).

We demonstrate that, by defining various combinations of these transformations, we can succinctly specify the precision of almost all existing pointer analysis algorithms.

Chapter 5

Conclusion

Pointer analysis is a fundamental enabling technology for program analysis. The goal of pointer analysis is to resolve the indirection, both in data-flow and control-flow, that is present in almost all programming languages. The more precisely that pointer analysis can resolve this indirection, the more effective the subsequent program analysis can be. Therefore, by improving the scalability of precise pointer analysis, we will make a positive impact across a wide range of program analyses used for many different purposes, including program verification, model checking, optimization, parallelization, program understanding, hardware synthesis, and more.

In this thesis, we have presented a suite of new algorithms aimed at improving pointer analysis scalability. We have focused specifically on two types of pointer analysis: inclusion-based flow- and context-insensitive pointer analysis and flow-sensitive, context-insensitive pointer analysis. These new algorithms make inclusion-based analysis over $4\times$ faster while using $7\times$ less memory than the previous state-of-the-art (Chapter 2); they also enable flow-sensitive pointer analysis to handle programs with millions of lines of code, two orders of magnitude greater than the previous state of the art (Chapter 3).

If we examine the entire set of algorithms described in this thesis, a common theme emerges: all of the algorithms are based on identifying and exploiting various types of *equivalence*. The four types of equivalence exploited are *pointer* equiva-

lence (cycle detection in Section 2.3, HVN, HR, HU, and HRU in Section 2.4.1, top-level pointer equivalence in Section 3.3.2.1), *location* equivalence in Section 2.4.2, *program-point* equivalence (sparseness in Sections 3.3, 3.4), and *access* equivalence in Section 3.4.1.3. We believe that these notions of equivalence apply to more than inclusion-based and flow-sensitive pointer analyses; they can be exploited for other types of pointer analysis as well, such as context-sensitive pointer analysis. Extending the application of these equivalences to new types of pointer analysis is an interesting direction for future work.

Another contribution of this thesis is a formal framework for describing the space of pointer analysis approximations. The space of possible approximations is complex and multi-dimensional, and until now has not been well-defined in a formal manner. We believe that the framework is useful for its own sake as a method to meaningfully compare the precision of the multitude of existing pointer analyses, as well as aiding in the systematic exploration of the entire space of approximations. In addition, such a formal framework is a necessary first step towards an even more ambitious goal: *declarative* pointer analysis. A given pointer analysis approximation could be specified using our formal framework, and an efficient, provably correct pointer analysis algorithm could be (semi-)automatically synthesized to compute the given approximation. While such a system is currently out of reach, it would greatly aid a systematic search of the space of approximations in order to find "sweet spots" in the trade-off between precision and performance, as well as being a great boon to non-pointer analysis experts who still need to derive correct and efficient pointer analysis algorithms to use for their own program analyses.

In summary, this thesis has presented a method to characterize the space of pointer analysis approximations and a set of new algorithms that significantly

extend scalability for two distinct points in this space. The formal framework and the principled approach used to create these new algorithms offer clear avenues for future work, as outlined above.

Bibliography

[1] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[2] Dzintars Avots, Michael Dalton, V. Benjamin Livshits, and Monica S. Lam. Improving software security with a c pointer analysis. In *27th International Conference on Software Engineering (ICSE)*, pages 332–341, 2005.

[3] John Aycock and R. Nigel Horspool. Simple generation of static single-assignment form. In *9th International Conference on Compiler Construction (CC)*, pages 110–124, 2000.

[4] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation (PLDI)*, pages 203–213, 2001.

[5] Rajeev Barua, Walter Lee, Saman Amarasinghe, and Anant Agarawal. Compiler support for scalable and efficient memory systems. *IEEE Trans. Comput.*, 50(11):1234–1247, 2001.

[6] William C. Benton and Charles N. Fischer. Interactive, scalable, declarative program analysis: from prototype to implementation. In *9th International Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 13–24, 2007.

[7] Marc Berndl, Ondrej Lhotak, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using BDDs. In *Programming Language Design and*

Implementation (PLDI), pages 103–114, 2003.

[8] Gianfranco Bilardi and Keshav Pingali. Algorithms for computing the static single assignment form. *Journal of the ACM*, 50(3):375–425, 2003.

[9] Glenn Bruns and Satish Chandra. Searching for points-to analysis. *SIGSOFT Software Engineering Notes*, 27(6):61–70, 2002.

[10] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEETC*, C-35(8):677–691, Aug 1986.

[11] J. Cai and R. Paige. Program derivation by fixed point computation. *Science of Computer Programming*, 11(3):197–261, 1989.

[12] Venkatesan T. Chakaravarthy. New results on the computability and complexity of points–to analysis. In *Symposium on Principles of Programming Languages (POPL)*, pages 115–125, 2003.

[13] Walter Chang, Brandon Streiff, and Calvin Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Computer and Communications Security (CCS)*, pages 39–50, 2008.

[14] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Programming Language Design and Implementation (PLDI)*, pages 296–310, 1990.

[15] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Symposium on Principles of Database Systems (PODS)*, pages 34–43, 1998.

[16] Peng-Sheng Chen, Ming-Yu Hung, Yuan-Shin Hwang, Roy Dz-Ching Ju, and Jenq Kuen Lee. Compiler support for speculative multithreading architecture with probabilistic points-to analysis. *SIGPLAN Notices*, 38(10):25–36, 2003.

[17] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Principles of Programming Languages (POPL)*, pages 232–245, 1993.

[18] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Symposium on Principles of Programming Languages (POPL)*, pages 55–66, 1991.

[19] Fred Chow, Sun Chan, Shin-Ming Liu, Raymond Lo, and Mark Streich. Effective representation of aliases and indirect memory operations in SSA form. In *Computational Complexity*, 1996.

[20] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[21] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977.

[22] R. Cytron and R. Gershbein. Efficient accomodation of may-alias information in SSA form. In *Programming Language Design and Implementation (PLDI)*, pages 36–45, June 1993.

[23] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[24] Ron K. Cytron and Jeanne Ferrante. Efficiently computing Φ-nodes on-the-fly. *ACM Transactions on Programming Languages and Systems*, 17(3):487–506, 1995.

[25] Manuvir Das. Unification-based pointer analysis with directional assignments. *ACM SIGPLAN Notices*, 35:535–46, 2000.

[26] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. *ACM SIGPLAN Notices*, 29(6):230–241, 1994.

[27] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Reducing the cost of data flow analysis by congruence partitioning. In *Computational Complexity*, pages 357–373, 1994.

[28] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Programming Language Design and Implementation (PLDI)*, pages 242–256, 1994.

[29] Manuel Faehndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. *ACM SIGPLAN Notices*, 33(5):85–96, 1998.

[30] Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. In *International Symposium on Software Testing and Analysis*, pages 133–144, 2006.

[31] Rakesh Ghiya. Putting pointer analysis to work. In *Principles of Programming Languages (POPL)*, pages 121–133, 1998.

[32] D. Goyal. An improved intra-procedural may-alias analysis algorithm. Technical report, New York University, 1999.

[33] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 108–124, 1997.

[34] Samuel Z. Guyer and Calvin Lin. Error checking with client-driven pointer analysis. *Science of Computer Programming*, 58(1-2):83–114, 2005.

[35] Brian Hackett and Radu Rugina. Region-based shape analysis with tracked locations. In *Symposium on Principles of Programming Languages (POPL)*, pages 310–323, 2005.

[36] B. Hardekopf, B. Wiedermann, W. Cook, and C. Lin. A unifying framework for describing the space of pointer analysis approximations. Technical Report TR-08-32, The University of Texas at Austin, 2008.

[37] Ben Hardekopf and Calvin Lin. The Ant and the Grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *Programming Language Design and Implementation (PLDI)*, pages 290–299, 2007.

[38] Ben Hardekopf and Calvin Lin. Exploiting pointer and location equivalence to optimize pointer analysis. In *International Static Analysis Symposium (SAS)*, pages 265–280, 2007.

[39] Ben Hardekopf and Calvin Lin. Semi-sparse flow-sensitive pointer analysis. In *Principles of Programming Languages (POPL)*, 2009.

[40] Rebecca Hasti and Susan Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *Programming Language Design and Implementation (PLDI)*, pages 97–105, 1998.

[41] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Programming Language Design and Implementation (PLDI)*, pages 23–34, 2001.

[42] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gr Egoire Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.

[43] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 54–61, 2001.

[44] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, 1999.

[45] Michael Hind and Anthony Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In *Static Analysis Symposium (SAS)*, pages 57–81, 1998.

[46] Susan Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Transactions on Programming Languages and Systems*, 19(1):1–6, 1997.

[47] Vineet Kahlon. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In *Programming language design and implementation*, pages 249–259, 2008.

[48] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:309–317, 1977.

[49] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, 1992.

[50] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Programming Language Design and Implementation (PLDI)*, pages 235–248, 1992.

[51] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *Programming Language Design and Implementation (PLDI)*, pages 24–31, 1988.

[52] Chris Lattner. LLVM: An infrastructure for multi-stage optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Dec 2002.

[53] O. Lhotak, S. Curial, and J.N. Amaral. Using ZBDDs in points-to analysis. In *Workshops on Languages and Compilers for Parallel Computing (LCPC)*, 2007.

[54] J. Lind-Nielson. BuDDy, a binary decision package.

[55] Thomas J. Marlowe, William G. Landi, Barbara G. Ryder, Jong-Deok Choi, Michael G. Burke, and Paul Carini. Pointer-induced aliasing: a clarification. *SIGPLAN Notices*, 28(9):67–70, 1993.

[56] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Precise and efficient call graph construction for c programs with function pointers. *Automated*

Software Engineering special issue on Source Code Analysis and Manipulation, 11(1):7–26, 2004.

[57] Ana Milanova and Barbara G. Ryder. Annotated inclusion constraints for precise flow analysis. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 187–196, 2005.

[58] M. Mock, D. Atkinson, C. Chambers, and S. Eggers. Improving program slicing with dynamic points-to data. In *Foundations of Software Engineering*, pages 71–80, 2002.

[59] Robert Muth and Saumya Debray. On the complexity of flow-sensitive dataflow analysis. Technical Report 99-12, Department of Computer Science, University of Arizona, 2000.

[60] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Computational Complexity*, pages 213–228, 2002.

[61] F. Nielson, H. R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[62] Diego Novillo. Design and implementation of Tree SSA, 2004.

[63] Esko Nuutila and Eljas Soisalon-Soininen. On finding the strong components in a directed graph. Technical Report TKO-B94, Helsinki University of Technology, Laboratory of Information Processing Science, 1995.

[64] Erik M. Nystrom, Hong-Seok Kim, and Wen mei W. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *International Symposium on Static Analysis*, pages 165–180, 2004.

[65] Dusko Pavlovic and Douglas R. Smith. Software development by refinement. In Bernhard K. Aichernig and Tom Maibaum, editors, *Formal Methods at the Crossroads*, volume 2757 of *Lecture Notes in Computer Science*. Springer Verlag, 2003.

[66] David Pearce, Paul Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis for C. In *ACM Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 37–42, 2004.

[67] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. Online cycle detection and difference propagation for pointer analysis. In *3rd International IEEE Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 3–12, 2003.

[68] G. Ramalingam. On sparse evaluation representations. *Theoretical Computer Science*, 277(1-2):119–147, 2002.

[69] John H. Reif and Harry R. Lewis. Symbolic evaluation and the global value graph. In *Principles of programming languages (POPL)*, pages 104–118, 1977.

[70] Atanas Rountev and Satish Chandra. Off-line variable substitution for scaling points-to analysis. *ACM SIGPLAN Notices*, 35(5):47–56, 2000.

[71] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.

[72] Alexandru Salcianu and Martin Rinard. Pointer and escape analysis for multithreaded programs. In *Symposium on Principles and Practices of Parallel Programming (PPoPP)*, pages 12–23, 2001.

[73] David A. Schmidt. Trace-based abstract interpretation of operational semantics. *Lisp Symbolic Computing*, 10(3):237–271, 1998.

[74] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, pages 201–220, 2001.

[75] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Symposium on Principles of Programming Languages (POPL)*, pages 1–14, 1997.

[76] M. Sharir and A. Pnueli. *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

[77] Manu Sridharan and Rastislav Bodik. Refinement-based context-sensitive points-to analysis for Java. *SIGPLAN Notices*, 41(6):387–400, 2006.

[78] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages (POPL)*, pages 32–41, 1996.

[79] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, June 1972.

[80] Teck Bok Tok, Samuel Z. Guyer, and Calvin Lin. Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers. In *15th International Conference on Compiler Construction (CC)*, pages 17–31, 2006.

[81] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis. In *Programming Language Design and Implementation (PLDI)*, pages 131–144, 2004.

[82] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Programming Language Design and Implementation (PLDI)*, pages 1–12, 1995.

[83] Suan Hsi Yong, Susan Horwitz, and Thomas W. Reps. Pointer analysis for programs with structures and casting. In *Programming Language Design and Implementation (PLDI)*, pages 91–103, 1999.

[84] Xin Zheng and Radu Rugina. Demand-driven alias analysis for C. In *Symposium on Principles of Programming Languages (POPL)*, pages 197–208, 2008.

[85] Jianwen Zhu. Symbolic pointer analysis. In *International Conference on Computer-Aided Design (ICCAD)*, pages 150–157, 2002.

[86] Jianwen Zhu. Towards scalable flow and context sensitive pointer analysis. In *Conference on Design Automation (DAC)*, pages 831–836, 2005.

[87] Jianwen Zhu and Silvian Calman. Symbolic pointer analysis revisited. In *Programming Language Design and Implementation (PLDI)*, pages 145–157, 2004.

Vita

Ben Hardekopf received a BSE in Electrical Engineering with a 2nd major in Computer Science from Duke University in 1997. While serving as an active duty officer in the United States Air Force, he received a Masters in Computer Science from SUNY at Utica/Rome in 2000. In 2001 he entered the Ph.D. program at The University of Texas at Austin.

Permanent address: 11005 Floral Park #2135
Austin, Texas 78759

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.