Semi-Sparse Flow-Sensitive Pointer Analysis

Ben Hardekopf Calvin Lin
The University of Texas at Austin
{benh,lin}@cs.utexas.edu

Abstract

Pointer analysis is a prerequisite for many program analyses, and the effectiveness of these analyses depends on the precision of the pointer information they receive. Two major axes of pointer analysis precision are *flow-sensitivity* and *context-sensitivity*, and while there has been significant recent progress regarding scalable context-sensitive pointer analysis, relatively little progress has been made in improving the scalability of flow-sensitive pointer analysis.

This paper presents a new interprocedural, flow-sensitive pointer analysis algorithm that combines two ideas—*semi-sparse* analysis and a novel use of BDDs—that arise from a careful understanding of the unique challenges that face flow-sensitive pointer analysis. We evaluate our algorithm on 12 C benchmarks ranging from 11K to 474K lines of code. Our fastest algorithm is on average 197× faster and uses 4.6× less memory than the state of the art, and it can analyze programs that are an order of magnitude larger than the previous state of the art.

Categories and Subject Descriptors D.3.4 [*Processors*]: Compilers; F.3.2 [*Semantics of Programming Languages*]: Program Analysis

General Terms Algorithms, Languages

Keywords Pointer analysis, alias analysis

1. Introduction

Almost all program analyses are more effective when given precise pointer information, and the scalability of such program analyses is often dictated by the precision of this pointer information [46]. Two major dimensions of pointer analysis precision are *flow-sensitivity* and *context-sensitivity*, which improve precision in complementary ways. A context-sensitive analysis respects the semantics of procedure calls by analyzing each distinct procedure context independently, whereas a context-insensitive analysis merges contexts together. A flow-sensitive analysis respects the control-flow of a program and instead computes a separate solution for each program point, whereas a flow-insensitive analysis does not respect control-flow and computes a single solution that conservatively holds for the entire program.

Recently, the scalability of both flow-insensitive pointer analysis [4, 22, 23, 25, 40, 41] and context-sensitive pointer analysis [10, 30, 33, 36, 39, 48, 52] has been greatly improved. In contrast, there has been relatively little progress on improving the performance of flow-sensitive pointer analysis. This lack of progress is unfortunate, because flow-sensitive pointer analysis has been shown to be beneficial to important problems, also known as *clients*, such as security analysis [7, 17], deep error checking [20], hardware synthesis [51], and the analysis of multi-threaded programs [45], among others [3, 9, 18].

In this paper, we present a new interprocedural, flow-sensitive pointer analysis algorithm that significantly improves upon the state of the art. The algorithm as presented is context-insensitive, but it could be extended to add context-sensitivity using one of several available techniques. Indeed, there is evidence that flow-sensitivity and context-sensitivity conspire to improve the behavior of client analyses [20]. Nevertheless, by concentrating exclusively on flow-sensitivity, we isolate its effects and directly address its particular challenges.

1.1 Challenges and Insights

Flow-sensitive pointer analysis presents unique challenges that hinder scalability. We will discuss these challenges in detail in Section 2.3 after describing flow-sensitive pointer analysis in Section 2.1, but we summarize the challenges now so that we can explain our insights for dealing with them.

Traditional flow-sensitive pointer analysis relies on the standard iterative dataflow technique, which must conservatively and inefficiently propagate pointer information to all reachable program points in case any of those points uses that information. Many program analyses have instead employed *static single assignment* (SSA) form to enable *sparse analysis*, which allows dataflow information to flow directly from variable definitions to their corresponding uses [43]. These def-use chains allow the analysis to avoid propagating information where it is not needed, greatly increasing analysis efficiency. Unfortunately, the construction of these def-use chains requires pointer analysis to determine where variables are defined and used, so pointer analysis itself is unable to exploit this technique.

In addition, flow-sensitive pointer analysis has prohibitive memory requirements and uses expensive set operations. Previous work on pointer analysis has applied *symbolic analysis*, using binary decision diagrams (BDDs) [6], to both reduce memory usage and decrease the cost of set operations [4, 48, 52]. Unfortunately, flow-sensitive analysis presents a challenge for symbolic analysis because of the presence of *strong updates*. Strong updates enable the analysis to kill old pointer information when a variable is assigned new information. Indirect strong updates (those involving memory store instructions such as `*x = y`) are problematic for symbolic analysis because they require that each program statement be processed independently, something for which BDDs are not well-suited. Previous attempts at using BDDs for flow-sensitive pointer analysis [52] have been forced to sacrifice precision to achieve acceptable performance.

Our algorithm overcomes these challenges with two insights:

POPL'09, January 18–24, 2009, Savannah, Georgia, USA.

1. There exists a subset of program variables (those that cannot be referenced indirectly via a pointer, called *top-level* variables) that can be converted to SSA form without the benefit of pointer information. For most programs, the majority of variables are top-level variables, so our solution employs a *partial* static single assignment form that performs a sparse analysis on top-level variables while using the standard iterative dataflow algorithm for the remaining variables.
2. We can both preserve the precision benefits of strong updates and obtain most of the performance benefit of symbolic analysis by confining the use of BDDs to the representation of pointer information, leaving the rest of the analysis unchanged. In effect, we use a partially-symbolic pointer analysis.

1.2 Contributions

This paper introduces a new flow-sensitive pointer analysis algorithm that is significantly more efficient—both in terms of analysis time and memory usage—than the previous state of the art. More specifically, our contributions are as follows:

- We identify three major challenges that limit the performance of flow-sensitive pointer analysis, and we explain how all previous approaches to optimizing the analysis—as well as our new algorithm—target one or more of these challenges.
- We introduce a new type of flow-sensitive pointer analysis called *semi-sparse analysis*, which significantly improves scalability. We further introduce two new optimizations for flow-sensitive analysis, *Top-Level Pointer Equivalence* and *Local Points-to Graph Equivalence*, that are enabled by the use of semi-sparse analysis.
- We present the first use of BDDs for flow-sensitive pointer analysis that allows the use of indirect strong updates. We also explore the BDD data structure's strengths and weaknesses compared to more conventional data structures.
- We compare our new semi-sparse analysis (SS) and semi-sparse analysis extended with our two new optimizations (SSO) against a baseline algorithm for flow-sensitive pointer analysis (IFS, standing for **I**terative dataflow **F**low-**S**ensitive analysis) based on the work of Hind and Pioli [28]. Our evaluation uses 12 C programs ranging in size from 11K to 474K lines of code (LOC), and it considers two different data structures for storing pointer information, namely, BDDs and sparse bitmaps. These two data structures are used to create two different versions of each algorithm: IFS, SS, and SSO.

  When using sparse bitmaps to store pointer information for both SSO and IFS, we find that SSO is 183× faster than IFS and uses 47× less memory. When using BDDs for both algorithms, we find that SSO is 114× faster than IFS and uses 1.4× less memory. Overall, our fastest analysis (SSO using BDDs) is 197× faster, uses 4.6× less memory, and can analyze programs that have 323K lines of code, which is an order of magnitude larger than the baseline algorithm (IFS using sparse bitmaps).

The rest of the paper is organized as follows. Section 2 explains flow-sensitive pointer analysis and identifies the three major challenges that hinder scalability. Section 3 describes related work and it addresses these challenges. Section 4 describes partial static single assignment form, Section 5 introduces our new semi-sparse analysis and optimizations, and Section 6 explores the usefulness of BDDs. Section 7 gives a detailed evaluation of all the analyses, and Section 8 concludes.

2. 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 [27].

2.1 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 L, a meet operator on the lattice, and a family of monotone transfer functions $f_i : L \rightarrow L$ 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:

$$\text{IN}_k = \bigcup_{x \in pred(k)} \text{OUT}_x \quad (1)$$

$$\text{OUT}_k = \text{GEN}_k \cup (\text{IN}_k - \text{KILL}_k) \quad (2)$$

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.

An important aspect of any pointer analysis is the *heap model*, i.e., how the conceptually infinite-size heap is abstracted into a finite set of memory locations. The most common practice, which we follow in this paper, is to treat each static memory allocation site as a single abstract memory location (which may map onto multiple concrete memory locations during program execution).

2.2 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 [28, 37], 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 [26]. 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 [7, 17], deep error checking [20], hardware synthesis [51], and the analysis of multi-threaded programs [45], among others [3, 9, 18].
- 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 [20]. 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 [8, 17, 20, 21, 49] 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.

2.3 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 (1) and (2).
2. **Expensive transfer functions.** Equations (1) and (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. In the next section we review past efforts at meeting these challenges before describing our own solution to the problem.

3. Related Work

The current state of the art for traditional flow-sensitive pointer analysis using iterative dataflow analysis is described by Hind and Pioli [27, 28], 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) [11, 16, 42].** 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.

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 [14] do 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 SSAs for the actual computation of pointer information.

Hasti and Horwitz [24] 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 [8] 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 [47]. The technique can analyze 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 [32], which automatically transforms a program into partial SSA form as described in Section 4. While the comparison is imperfect due to infrastructure differences, our fastest analysis (SSO using BDDs) 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 [2]. A number of papers have shown that symbolic analysis can greatly improve the performance of flow-insensitive pointer analysis [4, 48, 50, 52]. In addition, Zhu [51] 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 6. 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 particular client and applies flow-sensitive pointer analysis only to portions of the program that require that level of precision [20]. 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 [17]. 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 [29]. These approaches can be combined with our new flow-sensitive pointer analysis to achieve even greater scalability.

4. Partial Static Single Assignment Form

Static single assignment (SSA) form is an intermediate representation that requires each variable in a program to be defined exactly once. 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.

One benefit of SSA form is that each use of a variable is dominated by exactly one definition, so it is trivial to match definitions with their corresponding uses, enabling *sparse* analyses. Thus, SSA form addresses all three major challenges identified in Section 2.3: It speeds up convergence, reduces the number of times transfer functions need to be evaluated, and reduces the sizes of the points-to graphs stored at each node.

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

int*  w = &a;          w1 = ALLOC_a
int*  x = &b;          x1 = ALLOC_b
int** y = &c;          y1 = ALLOC_c
int** z =  y;          z1 = y1
      c =  0;          STORE 0 y1
     *y =  w;          STORE w1 y1
     *z =  x;          STORE x1 z1
      y = &d;          y2 = ALLOC_d
      z =  y;          z2 = y2
     *y =  w;          STORE w1 y2
     *z =  x;          STORE x1 z2
```

Figure 1. Example partial SSA code. On the left is the original C code, on the right is the transformed code in partial SSA form.

There are many known algorithms for converting a program into SSA form [1, 5, 13, 15]. However, the problem becomes more difficult when we consider indirect definitions through pointers. To correctly construct SSA form, we must know which variables are defined and/or used at each statement, which in turn requires pointer analysis. Even after pointer information becomes available, we must either greatly complicate the SSA form [12] or sacrifice much of its utility [31].

To overcome these issues, modern compilers such as GCC [38] and LLVM [32] use a variant of SSA, which we refer to as *partial* SSA form. The key idea is to divide variables into two classes. One class contains variables that are never referenced by pointers, so their definitions and uses can be trivially determined by inspection, 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, and these variables are not placed in SSA form because of the above-mentioned complications.

4.1 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 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.

```
int **a, *b, c;
      a = &b;          a = ALLOC_b
      b = &c;          t = ALLOC_c
      c = 0;           STORE t a
                       STORE 0 t
```

Figure 2. Example partial SSA code. On the left is the original C code, on the right is the transformed code in partial SSA form.

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 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 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 paper 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.

4.2 Advantages of Partial SSA

For flow-sensitive pointer analysis, partial SSA form has several important implications which 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.

5. Semi-Sparse Analysis

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.

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 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.

5.1 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, 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, shown in Table 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 it 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 [42]) 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 shows the DFG corresponding to the partial SSA code in Figure 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

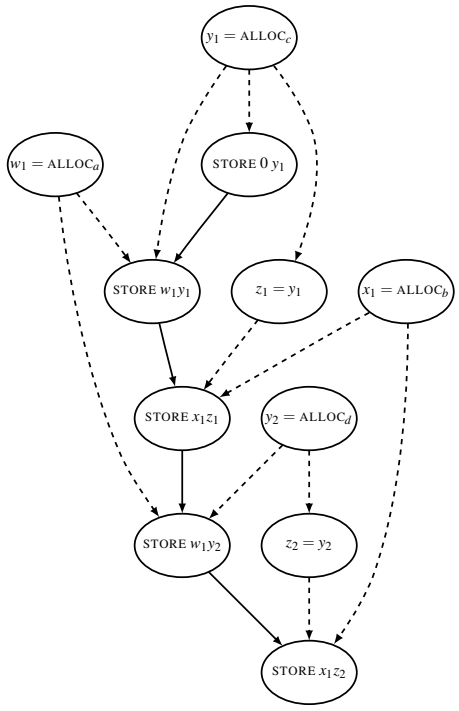

Figure 3. Example DFG corresponding to the code in Figure 1. Dashed edges are def-use chains; solid edges are for the SEG.

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.

□

5.2 The Analysis

The pointer analysis itself is similar to that described by Hind and Pioli [27, 28]. 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}).
- 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 $P_k(\mathtt{v})$ 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 $P_{top}(\mathtt{v})$ 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 1. 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 5–10. These algorithms use the helper functions listed in Algorithms 2–4. The $\hookleftarrow$ operator represents set update and $\xrightarrow{du}$ and $\xrightarrow{SEG}$ represent a def-use edge or SEG edge in the DFG, respectively.

Algorithm 1 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 2 propagateTopLevel(F, k)

  if PG_{top} changed **then**
    $StmtWorklist_F \hookleftarrow \{ n \mid k \xrightarrow{du} n \in E \}$

Algorithm 3 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 4 filter(k)

  return the subset of IN_k reachable from either a call argument or global variable

5.3 Optimizations

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

5.3.1 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, 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 $\forall \mathtt{x}, \mathtt{y}, \mathtt{z} \in \textit{Variables}$: $\mathtt{x} \bowtie \mathtt{y}$ iff $\mathtt{x} \rightarrow \mathtt{z} \Leftrightarrow \mathtt{y} \rightarrow \mathtt{z}$. 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`),

Algorithm 5 processAlloc(F,k) : [x = ALLOC$_i$]

$PG_{top} \hookleftarrow \{\mathtt{x} \rightarrow \text{ALLOC}_i\}$
propagateTopLevel(F,k)

Algorithm 6 processCopy(F,k) : [x = y z ...]

for all $v \in$ right-hand side **do**
  $PG_{top} \hookleftarrow \{\mathtt{x} \rightarrow P_{top}(v)\}$
propagateTopLevel(F,k)

Algorithm 7 processLoad(F,k) : [x = *y]

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

Algorithm 8 processStore(F,k) : [*x = y]

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

Algorithm 9 processCall(F,k) : [x = foo(y)]

if foo is a function pointer **then**
  targets := P_{top}(foo)
else
  targets := {foo}
filt := filter(k)
for all $C \in$ *targets* **do**
  for all call arguments a and corresponding parameters p **do**
    $PG_{top} \hookleftarrow \{\mathtt{p} \rightarrow P_{top}(\mathtt{a})\}$
    propagateTopLevel(C,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 10 processRet(F,k) : [return x]

callsites := the set of CALL statements targeting F
for all $n \in$ *callsites* **do**
  Let F_n be the function containing n
  $\text{OUT}_n \hookleftarrow \text{OUT}_k$
  propagateAddrTaken(F_n,n)
  if n is of the form r = F(...) **then**
    $PG_{top} \hookleftarrow \{\mathtt{r} \rightarrow P_{top}(\mathtt{x})\}$
    propagateTopLevel(F_n,n)
  if *StmtWorklist*$_{F_n}$ changed **then**
    FunctionWorklist $\hookleftarrow \{F_n\}$

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 [23, 44].

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 [2]. A number of papers have shown that symbolic analysis can greatly improve the performance of flow-insensitive pointer analysis [4, 48, 50, 52]. In addition, Zhu [51] 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. This work is explored in more detail in Section 6. Zhu does not show results for a flow-sensitive, context-insensitive analysis, so we cannot directly compare his techniques with ours. 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* $[x = y] \Rightarrow x \bowtie y$.

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 $\mathtt{x} \bowtie \mathtt{y}$. □

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:

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

In Figure 3, $\mathtt{y}_1 \bowtie \mathtt{z}_1$ and $\mathtt{y}_2 \bowtie \mathtt{z}_2$. We can replace all occurrences of $\mathtt{z}_1$ with $\mathtt{y}_1$, replace all occurrences of $\mathtt{z}_2$ with $\mathtt{y}_2$, and eliminate the nodes for $[\mathtt{z}_1 = \mathtt{y}_1]$ and $[\mathtt{z}_2 = \mathtt{y}_2]$. The def-use edge from $[\mathtt{y}_1 = \text{ALLOC}_c]$ to $[\mathtt{z}_1 = \mathtt{y}_1]$ is removed, and a new def-use edge is added from $[\mathtt{y}_1 = \text{ALLOC}_c]$ to $[\text{STORE } \mathtt{x}_1\ \mathtt{y}_1]$. Similarly, the def-use edge from $[\mathtt{y}_2 = \text{ALLOC}_d]$ to $[\mathtt{z}_2 = \mathtt{y}_2]$ is removed, and a new def-use edge is added from $[\mathtt{y}_2 = \text{ALLOC}_d]$ to $[\text{STORE } \mathtt{x}_1\ \mathtt{y}_2]$. Figure 4 shows the optimized version of Figure 3.

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

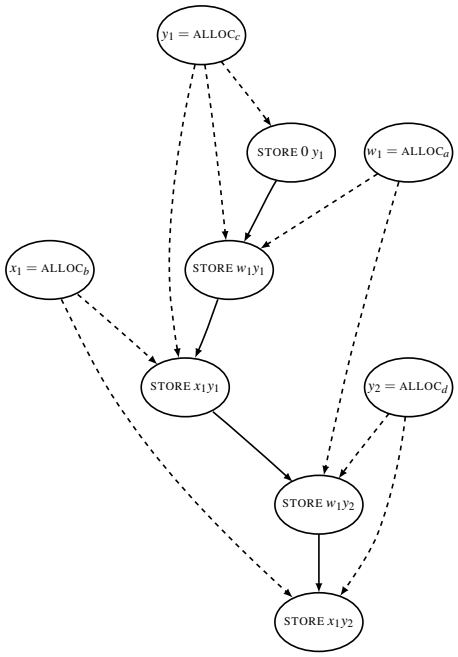

Figure 4. Figure 3 optimized using top-level pointer equivalence.

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 $\mathtt{x} = \mathtt{y} \in S$, then $\mathtt{x}, \mathtt{y} \in V$ 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. □

5.3.2 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 reduce 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 11) 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 11 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 $typeof(k) \in \{\text{STORE}, \text{CALL}\}$ **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 11 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 7.

6. 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 [6].

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 [4, 48, 50, 51, 52]. 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 [51], 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 compute 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 operations 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 [28]) and our new semi-sparse analysis.

7. Experimental Evaluation

To evaluate our new techniques, we implement three flow-sensitive pointer analysis algorithms: a baseline analysis based on Hind and Pioli [28] (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) and for each algorithm we evaluate two versions, one that implements pointer information using sparse bitmaps and a second that uses BDDs.

The bitmap versions of IFS, SS, and SSO filter pointer information at call-sites as described by Hind and Pioli (see Section 3 and Section 5.2). 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 the use of 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 algorithms are implemented in the LLVM compiler infrastructure [32], and the BDDs use the BuDDy BDD library [35]. The algorithms are written in C++ and handle all aspects of the C language except for varargs. The source code for the various algorithms is freely available at the authors' website.

The benchmarks for our experiments are described in Table 2. Six of the benchmarks are taken from SPECINT 2000 (the largest six applications from that suite) and six from various open-source applications. 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.

7.1 Performance Results

Table 3 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.

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` 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 5 and 6 describe various analysis statistics to explain the relative performance of these algorithms. Figure 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 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 1).

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.

Name	Description	LOC	Statements	Functions	Call Sites
197.parser	parser	11.4K	33.6K	99	774
ex-050325	text processor	34.4K	37.0K	325	2,519
300.twolf	place and route simulator	20.5K	45.0K	107	331
255.vortex	object-oriented database	67.2K	69.2K	271	4,420
sendmail-8.11.6	email server	88.0K	69.3K	273	3,203
254.gap	group theory interpreter	71.4K	132.2K	725	6,002
253.perlbmk	PERL language	85.5K	184.6K	726	8,597
vim-7.1	text processor	323.5K	316.4K	1,935	15,962
nethack-3.4.3	text-based game	252.6K	356.3K	1,385	23,001
176.gcc	C language compiler	226.5K	376.2K	1,159	19,964
gdb-6.7.1	debugger	474.1K	484.3K	3,801	37,119
ghostscript-8.15	postscript viewer	429.0K	494.0K	4,815	18,050

Table 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.

Name	*bitmap*						*BDD*					
	IFS		SS		SSO		IFS		SS		SSO	
	time	mem	time	mem	time	mem	time	mem	time	mem	time	mem
197.parser	80.25	888	1.28	53	0.52	15	7.24	142	0.64	142	0.48	142
ex-050325	—	OOM	15.74	198	7.33	39	7.95	142	0.66	143	0.46	142
300.twolf	72.28	415	0.82	32	0.34	12	6.41	143	0.46	144	0.32	143
255.vortex	—	OOM	33.37	1,275	11.70	81	14.39	150	0.97	151	0.78	150
sendmail-8.11.6	—	OOM	—	OOM	86.38	258	38.51	150	2.16	154	1.40	152
254.gap	—	OOM	—	OOM	191.72	518	68.66	167	2.50	168	2.34	166
253.perlbmk	—	OOM	—	OOM	—	OOM	1,477.05	280	50.22	182	21.25	177
vim-7.1	—	OOM	—	OOM	—	OOM	4,759.37	535	573.28	300	112.16	263
nethack-3.4.3	—	OOM	—	OOM	4,762.07	1,648	3,435.48	423	13.68	225	5.37	220
176.gcc	—	OOM	—	OOM	—	OOM	2,445.27	595	39.71	234	9.37	226
gdb-6.7.1	—	OOM	—	OOM	—	OOM	OOT	—	OOT	—	OOT	—
ghostscript-8.15	—	OOM	—	OOM	—	OOM	OOT	—	OOT	—	OOT	—

Table 3. Performance: time (in seconds) and memory consumption (in megabytes) of the various analyses. Results under the *bitmap* columns are obtained using pointer information implemented using sparse bitmaps; those under the *BDD* columns are obtained using pointer information implemented using BDDs. OOM means the benchmark ran out of memory; OOT means it ran out of time (exceeded an eight hour time limit).

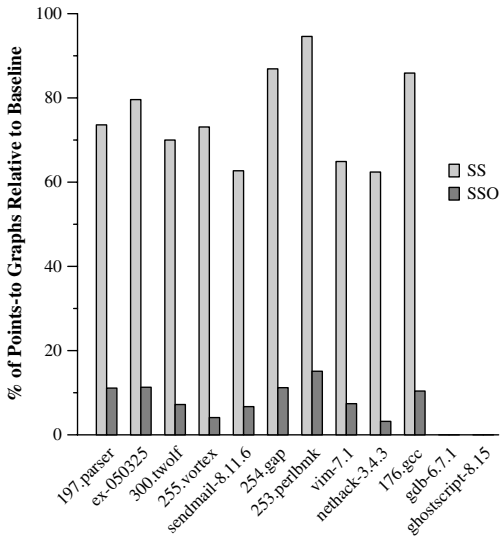

Figure 5. Number of points-to graphs maintained by SS and SSO compared to IFS. Lower is better (fewer points-to graphs).

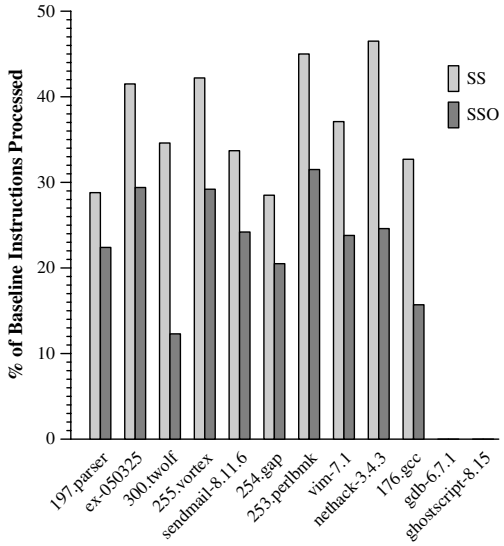

Figure 6. Number of instructions processed by SS and SSO compared to IFS. Lower is better (fewer instructions processed).

7.2 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 before the analysis begins, then expanded as necessary. As we look at 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 [34, 48]

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 [8, 47] 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 [19]. 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.

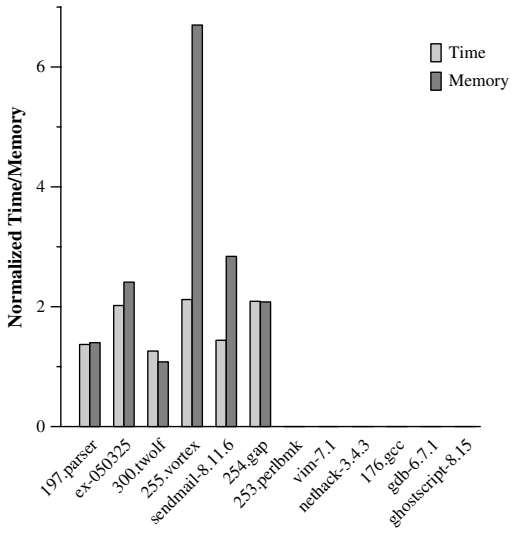

Figure 7. Analysis time and memory usage (normalized to our baseline) for the bitmap version of SSO *without* the assumption on CALLs versus SSO *with* the assumption—i.e., $\text{SSO}_{without}/\text{SSO}_{with}$.

7.3 SSO Precision

The version of SSO used in these experiments makes use of the assumption discussed at the end of Section 5.3.2, 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 7 and 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%.

8. Conclusion

Flow-sensitive pointer analysis is an important enabling technology for program analysis. We have identified the major challenges that stand in the way of scalable flow-sensitive pointer analysis, and we have directly addressed these challenges with our new *semi-sparse analysis*, thereby significantly improving on the previous state of

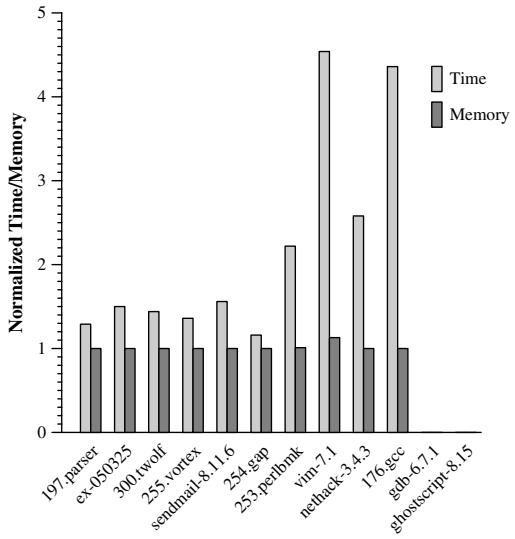

Figure 8. Analysis time and memory usage (normalized to our baseline) for the BDD version of SSO *without* the assumption on CALLs versus SSO *with* the assumption—i.e., $\text{SSO}_{without}/\text{SSO}_{with}$.

the art. We have also described how BDDs can be effectively used for a fully flow-sensitive pointer analysis without sacrificing precision. Our techniques are 197× faster and use 4.6× less memory than traditional flow-sensitive pointer analysis.

In the future we plan on further optimizing the analysis, implementing a number of precision-enhancing features, and building various client analyses (such as security and error-checking applications) to showcase the usefulness of our techniques. We believe that flow-sensitive pointer analysis has an important position in the realm of program analysis and that our work has made it possible for clients that use flow-sensitive pointer information to scale to applications with hundreds of thousands of lines of code.

Acknowledgments

We thank Kathryn McKinley, Sam Guyer, Michael Hind, and the anonymous reviewers for their helpful comments on early versions of this paper. This research was supported by Air Force Research Laboratory contract FA8750-07-C-0035 from the Disruptive Technology Office and from a grant from the Intel Research Council.

References

[1] J. Aycock and R. N. Horspool. Simple generation of static single-assignment form. In *9th International Conference on Compiler Construction (CC)*, pages 110–124, London, UK, 2000. Springer-Verlag.

[2] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. In *Programming Language Design and Implementation (PLDI)*, pages 203–213, 2001.

[3] R. Barua, W. Lee, S. Amarasinghe, and A. Agarawal. Compiler support for scalable and efficient memory systems. *IEEE Trans. Comput.*, 50(11):1234–1247, 2001.

[4] M. Berndl, O. Lhotak, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *Programming Language Design and Implementation (PLDI)*, 2003, pages 103–114.

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

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

[7] W. Chang, B. Streiff, and C. Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Computer and Communications Security (CCS)*, 2008, pages 39–50.

[8] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Programming Language Design and Implementation (PLDI)*, pages 296–310, 1990.

[9] P.-S. Chen, M.-Y. Hung, Y.-S. Hwang, R. D.-C. Ju, and J. K. Lee. Compiler support for speculative multithreading architecture with probabilistic points-to analysis. *SIGPLAN Not.*, 38(10):25–36, 2003.

[10] B.-C. Cheng and W.-M. W. Hwu. Modular interprocedural pointer analysis using access paths: Design, implementation, and evaluation. *ACM SIG-PLAN Notices*, 35(5):57–69, 2000.

[11] J.-D. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Symposium on Principles of Programming Languages (POPL)*, pages 55–66, New York, NY, USA, 1991. ACM Press.

[12] F. Chow, S. Chan, S.-M. Liu, R. Lo, and M. Streich. Effective representation of aliases and indirect memory operations in SSA form. In *Compiler Construction*, 1996, pages 253–267.

[13] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

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

[15] R. K. Cytron and J. Ferrante. Efficiently computing Φ-nodes on-the-fly. *ACM Trans. Program. Lang. Syst*, 17(3):487–506, 1995.

[16] E. Duesterwald, R. Gupta, and M. L. Soffa. Reducing the cost of data flow analysis by congruence partitioning. In *Compiler Construction*, 1994, pages 357–373.

[17] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In *International Symposium on Software Testing and Analysis*, pages 133–144, 2006.

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

[19] D. Goyal. An improved intra-procedural may-alias analysis algorithm. Technical report TR1999-777, New York University, 1999.

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

[21] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *Symposium on Principles of Programming Languages*, pages 310–323, 2005.

[22] B. Hardekopf and C. 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, San Diego, CA, USA, 2007.

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

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

[25] N. Heintze and O. 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.

[26] M. Hind. Pointer analysis: haven't we solved this problem yet? In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 54–61, New York, NY, USA, 2001. ACM Press.

[27] M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer

alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, 1999.

[28] M. Hind and A. Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In *Static Analysis Symposium*, pages 57–81, 1998.

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

[30] H.-S. Kim, E. M. Nystrom, R. D. Barnes, and W.-M. W. Hwu. Compaction algorithm for precise modular context-sensitive points-to analysis. Technical report IMPACT-03-03, Center for Reliable and High Performance Computing, University of Illinois, Urbana-Champaign, 2003.

[31] C. Lapkowski and L. J. Hendren. Extended SSA numbering: introducing SSA properties to languages with multi-level pointers. In *CASCON '96: Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative research*, page 23, 1996.

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

[33] C. Lattner and V. Adve. Data structure analysis: An efficient context-sensitive heap analysis. Technical Report UIUCDCS-R-2003-2340, Computer Science Dept, University of Illinois at Urbana-Champaign, 2003.

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

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

[36] A. Milanova and B. 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.

[37] 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.

[38] D. Novillo. Design and implementation of Tree SSA, 2004.

[39] E. M. Nystrom, H.-S. Kim, and W. mei W. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *International Symposium on Static Analysis*, pages 165–180, 2004.

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

[41] D. J. Pearce, P. H. J. Kelly, and C. 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.

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

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

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

[45] A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. In *PPoPP '01: Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 12–23, 2001.

[46] M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. *Lecture Notes in Computer Science*, 1302:16–34, 1997.

[47] T. B. Tok, S. Z. Guyer, and C. 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.

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

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

[50] J. Zhu. Symbolic pointer analysis. In *International Conference on Computer-Aided Design (ICCAD)*, pages 150–157, New York, NY, USA, 2002. ACM Press.

[51] J. Zhu. Towards scalable flow and context sensitive pointer analysis. In *DAC '05: Proceedings of the 42nd Annual Conference on Design Automation*, pages 831–836, 2005.

[52] J. Zhu and S. Calman. Symbolic pointer analysis revisited. In *Programming Language Design and Implementation (PLDI)*, pages 145–157, New York, NY, USA, 2004. ACM Press.