Flow-Sensitive Pointer Analysis for Millions of Lines of Code

Ben Hardekopf
University of California, Santa Barbara
benh@cs.ucsb.edu

Calvin Lin
The University of Texas at Austin
lin@cs.utexas.edu

***Abstract*—Many program analyses benefit, both in precision and performance, from precise pointer analysis. An important dimension of pointer analysis precision is flow-sensitivity, which has been shown to be useful for applications such as program verification and static analysis of binary code, among many others. However, flow-sensitive pointer analysis has historically been unable to scale to programs with millions of lines of code.**

We present a new flow-sensitive pointer analysis algorithm that is an order of magnitude faster than the existing state of the art, enabling for the first time flow-sensitive pointer analysis for programs with millions of lines of code. Our flow-sensitive algorithm is based on a sparse representation of program code created by a staged, flow-insensitive pointer analysis. We explain how this new algorithm is a member of a new family of pointer analysis algorithms that deserves further study.

I. Introduction

Pointer analysis is an important enabling technology that can improve the precision and performance of many program analyses by providing precise pointer information. Considerable progress has been made in various dimensions of pointer analysis, particularly with regard to flow-insensitive analysis [15], [16] and BDD-based context-sensitive pointer analysis [2], [32]–[34]. However, *flow-sensitive* pointer analysis has received relatively little attention, which is unfortunate because it has been shown to be important for a growing list of program analyses [6], [13], including those that check for security vulnerabilities [4], [12], [14], those that synthesize hardware [34], and those that analyze multi-threaded codes [29].

One reason for this lack of attention may be the special challenges that flow-sensitive pointer analysis presents. Unlike a flow-insensitive analysis, which ignores statement ordering and computes a single solution that holds for all program points, a flow-sensitive analysis respects a program's control-flow and computes a separate solution for each program point. Thus, the traditional flow-sensitive approach uses an iterative dataflow analysis (IDFA), which is extremely inefficient for pointer analysis. IDFA conservatively propagates all dataflow information from each node in the control-flow graph to every other reachable node, because the analysis cannot know which nodes might need that information. For large programs, the control flow graph (CFG) can have hundreds of thousands of nodes, with each node maintaining two points-to graphs—one for incoming information and one for outgoing information; each points-to graph can have hundreds of thousands of pointers; and each pointer can have thousands of elements in its points-to set. Thus, each node stores, propagates, and computes transfer functions on an enormous amount of information, which is inefficient in both time and space.

The typical method for optimizing a flow-sensitive dataflow analysis is to perform a *sparse analysis* [5], [27], which directly connects variable definitions (defs) with their uses, allowing data flow facts to be propagated only to those program locations that need the values. Unfortunately, sparse pointer analysis is problematic because pointer information is required to compute the very def-use information that would enable a sparse analysis. This paper shows how this difficulty can be overcome and how the use of a sparse analysis greatly increases the scalability of flow-sensitive pointer analysis.

A. Insights

The key insight behind our technique is to *stage* the pointer analysis. An *auxiliary* pointer analysis first computes conservative def-use information, which then enables the *primary* flow-sensitive analysis to be performed sparsely using the conservative def-use information. This idea actually defines a family of staged flow-sensitive analyses, described in Section IV-A, in which each member of the family uses a different auxiliary analysis (referred to henceforth as AUX). While other work on pointer analysis has previously employed auxiliary analyses [21], [28], none has used the results of the auxiliary analysis in the same way.

This paper explains how we address three main challenges facing a staged analysis, resulting in a scalable flow-sensitive pointer analysis. First, AUX must strike a good balance between precision and performance: If the results are too imprecise, the sparsity of the primary analysis will suffer; if AUX is not scalable, the primary analysis will never get to execute. Second, the primary analysis must deal with the complexity of incorporating the auxiliary analysis' conservative def-use information while maintaining the precision of a flow-sensitive pointer analysis. Third, the primary analysis must efficiently manage the extremely large number of def-use chains that are computed by AUX.

B. Contributions

This paper makes the following contributions:

- We present *staged flow-sensitive analysis (SFS),* a new flow-sensitive pointer analysis that introduces the notion

of staging, in which the def-use chains created by a less precise auxiliary pointer analysis are used to enable the sparsity of the primary flow-sensitive pointer analysis.
- We introduce the notion of *access equivalence*, which partitions def-use chains into equivalence classes, allowing SFS to efficiently process large amounts of def-use information.
- We evaluate an instance of SFS that uses inclusion-based pointer analysis as the auxiliary analysis. This particular auxiliary analysis is attractive because it is the most precise of the flow- and context-insensitive pointer analyses and because it can analyze millions of lines of code in a matter of minutes [15], [16].
- Using a collection of 16 open source C programs, we compare the SFS algorithm to a baseline, semi-sparse analysis, which represents the prior state-of-the-art [17]. We show that our staged algorithm is an order of magnitude more scalable than the baseline; for example, SFS is able to analyze a program with 1.9M LOC in under 14 minutes. The primary strength of SFS is its improved scalability. For small programs—those with fewer than 100K LOC—SFS is efficient but provides no performance advantage over the baseline. For mid-sized programs—those with 100K to 400K LOC—SFS is 5.5× faster than the baseline. For large programs—those with more than 800K LOC—SFS completes while the baseline does not.

The remainder of the paper is organized as follows. Section II gives background information that is important to understanding flow-sensitive pointer analysis and our technique. Section III discusses related work, Section IV describes our staging technique, and Section V gives a detailed description of the algorithm. Section VI experimentally evaluates the scalability of our technique compared to the current state of the art, and Section VII concludes.

II. Background

This section provides background that is needed for understanding the remainder of the paper. We describe the basics of flow-sensitive pointer analysis and static single assignment form, and then we describe LLVM, the particular compiler infrastructure used for our work.

A. Flow-Sensitive Pointer Analysis

Traditional flow-sensitive pointer analysis analysis is carried out on the *control-flow graph* (CFG), which is a directed graph, $G = \langle N, E \rangle$, where N is a finite set of nodes (or *program points*) corresponding to program statements and where $E \subseteq N \times N$ is a set of edges corresponding to the control-flow between statements. To ensure the decidability of the analysis, branch conditions are uninterpreted; thus branches are treated as non-deterministic.

Each node k of the CFG maintains two points-to graphs (i.e., sets of points-to relations): $\textsc{in}_k$ represents the incoming pointer information, and $\textsc{out}_k$ represents the outgoing pointer information. Each node has a transfer function that transforms $\textsc{in}_k$ to $\textsc{out}_k$; the function is characterized by the sets $\textsc{gen}_k$ and $\textsc{kill}_k$ 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. For all nodes k, the analysis iteratively computes the following two functions until convergence:

$$\textsc{in}_k = \bigcup_{x \in pred(k)} \textsc{out}_x \quad (1)$$

$$\textsc{out}_k = \textsc{gen}_k \cup (\textsc{in}_k - \textsc{kill}_k) \quad (2)$$

For a program statement k with an assignment `x = y`, $\textsc{kill}_k = \{\texttt{x} \to \_\}$, which is called a *strong update*: All relations with `x` on the left-hand side are removed and replaced with new relations. For a program statement k with an assignment `*x = y`, the matter is more complicated. If `x` definitely points to a single concrete memory location `z`, then $\textsc{kill}_k = \{\texttt{z} \to \_\}$ and the analysis still performs a strong update. However, if `x` may point to multiple concrete memory locations, then $\textsc{kill}_k = \{\}$ and the analysis performs a *weak update*: The analysis cannot be sure *which* memory location will be updated by the assignment, so it adds the new information but conservatively preserves all existing points-to relations. A pointer may refer to multiple concrete memory locations if its points-to set contains multiple locations or if the single location in its points-to set is either (1) a heap variable (as described in the next paragraph), or (2) a local variable of a recursive function (which may have multiple instances on the program's runtime stack at the same time).

An important aspect of any pointer analysis is the *heap model*, which abstracts the conceptually infinite-size heap into a finite set of memory locations. We adopt the common practice of treating each static memory allocation site as a distinct abstract memory location (which may map to multiple concrete memory locations during program execution).

B. SSA

In *static single assignment* (SSA) form [9], each variable is defined exactly once in the program text. Variables with multiple definitions in the original program are split into separate instances, one for each definition. At join points in the CFG, separate instances of the same variable are combined using a ϕ function, producing an assignment to a new instance of the variable. SSA form is ideal for performing sparse analyses because it explicitly represents def-use information and allows dataflow information to flow directly from variable definitions to their corresponding uses [27].

The conversion to SSA form is complicated by the existence of indirect defs and uses through pointers, which can only be discovered using pointer analysis. Because the resulting pointer information is conservative, each indirect def and use is actually a *possible* def or use. Following Chow et al [8], we use χ and μ functions to represent these possible defs and uses. Each indirect store (e.g., `*x = y`) in the original program representation is annotated with a function $v = \chi(v)$ for each variable v that may be defined by the store; similarly, each indirect load (e.g., `x = *y`) 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 def and use of the given variable, and each μ function is treated as a use of the given variable.

To avoid the complications of dealing with indirect loads and stores, some modern compilers such as GCC [26] and LLVM [23] use a variant of SSA that we call *partial* SSA form. The idea is to divide variables into two classes. One class contains *top-level variables* that are never referenced by pointers, so their definitions and uses can be trivially determined by inspection without pointer information. These variables can be converted to SSA using any algorithm for constructing SSA form. The second class contains those variables that *can* be referenced by pointers (*address-taken variables*), and to avoid the above-mentioned complications, these variables are not placed in SSA form.

C. The LLVM IR

Our implementation uses the LLVM compiler infrastructure, so to make our discussion concrete, we now describe LLVM's internal representation (IR) and its particular instantiation of partial SSA form. While the details are specific to LLVM, the ideas can be translated to other forms of partial SSA. In LLVM, 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 rather than registers, and they are not in SSA form. Top-level variables are modified using ALLOC (memory allocation) and COPY instructions. Address-taken variables are accessed via LOAD and STORE instructions, which take top-level pointer variables as arguments. The 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 (source statements with multiple levels of indirection are reduced to this form by introducing temporary variables).

Figure 1 shows 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. The rest of this paper will assume the use of the LLVM IR, which means that address-taken variables can only be defined or used by LOAD and STORE instructions.

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

`int*  w = &a;`	w_1 = ALLOC$_a$
`int*  x = &b;`	x_1 = ALLOC$_b$
`int** y = &c;`	y_1 = ALLOC$_c$
`int** z =  y;`	$z_1 = y_1$
`      c =  0;`	STORE 0 y_1
`     *y =  w;`	STORE w_1 y_1
`     *z =  x;`	STORE x_1 z_1
`      y = &d;`	y_2 = ALLOC$_d$
`      z =  y;`	$z_2 = y_2$
`     *y =  w;`	STORE w_1 y_2
`     *z =  x;`	STORE x_1 z_2

Fig. 1. An example of partial SSA form. On the left is the original C code; on the right is the transformed code in partial SSA form. Note that the address-taken variables `a`, `b`, `c`, and `d` are not mentioned syntactically in the transformed program text; they are referred to only indirectly via the top-level pointers `w`, `x`, and `y`.

III. Related Work

Most of the previous advancements in flow-sensitive pointer analysis have exploited some form of sparsity to improve performance.

Chase et al [5] propose that SSA form be dynamically computed during the course of the flow-sensitive pointer analysis. Chase et al do not evaluate their idea, but a similar idea is evaluated by Tok et al [31], whose algorithm can analyze C programs with almost 70,000 lines of code in roughly 30 minutes. The scalability of this approach is limited by the cost of dynamically updating SSA form. In contrast, our analysis pre-computes the SSA information prior to the actual flow-sensitive analysis.

Hardekopf and Lin [17] present a *semi-sparse* flow-sensitive pointer analysis, which exploits partial SSA form to perform a sparse analysis on top-level variables, while using iterative dataflow analysis on address-taken variables. By contrast, our staged analysis is completely sparse and uses SSA form for all variables. Our resulting analysis is an order of magnitude more scalable than semi-sparse analysis, as shown in Section VI.

Hasti and Horwitz [18] 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 fully flow-sensitive pointer analysis. No experimental evaluation of this technique has been published. In contrast to their work, which uses multiple passes to compute results that are not fully flow-sensitive, our staged algorithm uses a single pass of the flow-insensitive analysis to compute a fully precise flow-sensitive analysis.

Hind and Pioli [19], [20] use a weaker form of sparsity in the form of the *sparse evaluation graph* (SEG) [7], which is a graph that is derived from the CFG by eliding nodes that are irrelevant to pointer analysis (i.e., they do not manipulate pointer information) while at the same time maintaining the control-flow relations among the remaining nodes. Hind and Pioli [19], [20] also introduce the notion of *filtered forward binding*, which recognizes that when passing pointer information to the target of a function call, it is only necessary to pass pointer information that the callee can access from a global variable or from one of the function parameters. Hind and Pioli's paper evaluates benchmarks up to 30K LOC; a modern re-implementation of their algorithm evaluated by Hardekopf and Lin [17] was unable to do any better.

A staging strategy for flow-sensitive pointer analysis that builds on HSSA form [8] was apparently implemented in the Open64 compiler in 1996, though the implementation is solely intra-procedural and, to our knowledge, a description has never been published.

Previous work has also used notions similar to staging to improve the efficiency of pointer analysis. Client-driven pointer analysis [14] analyzes the needs of a particular client and applies flow-sensitive and context-sensitive pointer analysis only to those portions of the program that require that level of precision. Fink et al [12] apply a similar technique specifically for typestate analysis by successively applying a series of increasingly precise pointer analyses to successively smaller programs by pruning away portions of the program once they have been successfully verified. Ruf [28] and Kahlon [21] bootstrap the flow-sensitive pointer analysis by using a flow-insensitive pointer analysis to first partition the program into sections that can be analyzed independently. Unlike staging, these techniques use a preliminary analysis to reduce the size of the input by either pruning [12], [14] or partitioning [21] the program. By contrast, our staging employs the def-use chains computed by the auxiliary pointer analysis to help create more precise def-use information that in turn allows the algorithm to produce more precise pointer information. Thus, pruning and partitioning are orthogonal to staging, and the ideas can be combined: A staged analysis does not need to compute flow-sensitive solutions for all variables, only the relevant ones.

IV. Staging the Analysis

A key component of our new algorithm is AUX, the auxiliary pointer analysis that computes conservative def-use information for the primary flow-sensitive pointer analysis. The remainder of this section discusses AUX and explains how its results can be used to optimize the primary analysis.

A. *The Family of Staged Analyses*

As mentioned in the introduction, staged analysis represents a family of analyses, with each family member distinguished by the particular auxiliary pointer analysis that is used. As with any pointer analysis, there are two concerns: precision and scalability. A more precise AUX will yield a sparser primary analysis; a less scalable AUX could limit the scalability of the staged analysis.

The most natural choice for AUX is a flow- and context-insensitive analysis; there are many of these 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 [30], to Das' One-Level Flow [11], to inclusion-based (i.e., Andersen-style) analysis. Any of these choices will result in an overall staged analysis that provides the same precision as a traditional flow-sensitive analysis. Interestingly, Hardekopf and Lin's semi-sparse pointer analysis can be viewed as a bastardized member of the family, where AUX identifies top-level variables to allow the primary analysis to be partially sparse. For this paper we use an inclusion-based auxiliary analysis, which is the most precise of the flow- and context-insensitive pointer analyses, and which scales to millions of LOC [15], [16].

An intriguing choice for AUX, which we leave for future work, would be to use an analysis whose precision is incomparable to the primary analysis,[1] because by sequentially combining incomparable analyses we get some of the benefit of combining the two analyses but at lower cost. For example, by using a flow-insensitive, context-sensitive analysis for AUX, the overall analysis would incorporate both flow- and context-sensitive information, making it more precise than a traditional flow-sensitive, context-insensitive analysis. Such an analysis would be interesting for two reasons. First, full flow- and context-sensitivity often provide more precision than is necessary [14]. In fact, others have sacrificed some degree of precision (eg, disallowing indirect strong updates [34]) for the sake of improved scalability. Second, flow- and context-sensitivity conspire to limit scalability; by providing a way to perform the analyses in sequence rather than at the same time, staged analysis can yield a much more scalable algorithm. Of course, the precision of the resulting analysis would need to be carefully studied.

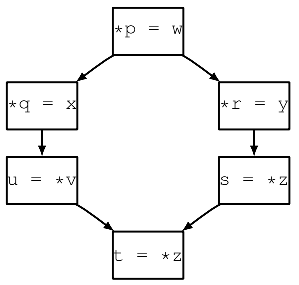
To be clear, as long as the AUX analysis is sound, the primary SFS analysis will also be sound and it will be at least as precise as a traditional flow-sensitive pointer analysis. The precision of AUX mainly affects the sparsity (and hence performance) of the primary analysis.

B. *Sparse Flow-Sensitive Pointer Analysis*

The primary data structure for SFS is a def-use graph (DUG), which contains a node for each program statement and which contains edges to 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 these def-use edges are computed, as well as how the precision of the flow-sensitive analysis is maintained 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. Once the LOADs and

[1]Inclusion-based analysis is not incomparable to the primary analysis—it is strictly less precise than a flow-sensitive pointer analysis.



AUX *points-to sets*

p → {a}
q → {b,c,d,e,f}
v → {e,f}
r → {a,b,d}
z → {a,b,c,d}

Fig. 2. An example CFG along with some hypothetical points-to sets that might be computed by AUX. These points-to sets will be used to construct the SSA representation of the program, shown in Figure 3.

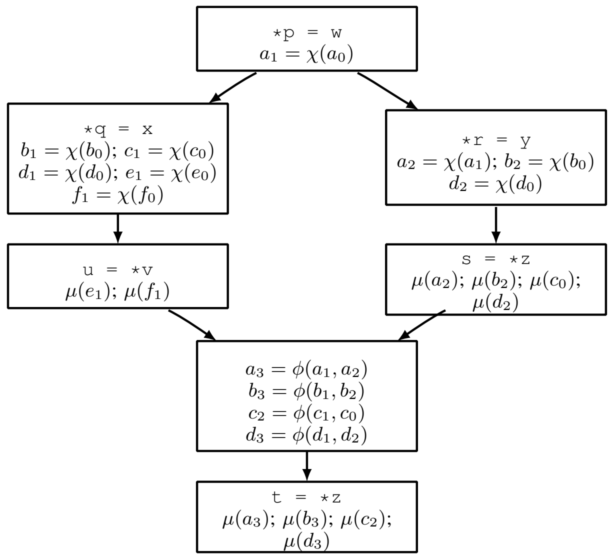


Fig. 3. The SSA representation for the CFG in Figure 2. The χ functions represent address-taken variables that might be defined by a store; the μ functions represent address-taken variables that might be accessed by a load.

STOREs are annotated with χ and μ functions as described in Section II-B, the program can be converted to SSA form using any standard SSA algorithm [1], [3], [9], [10]. Figure 2 shows an example program fragment along with some pointer information computed by AUX. Figure 3 shows the same program fragment annotated with χ and μ functions and translated into SSA. The annotation step comes directly from the points-to information computed by AUX—for example, for the CFG node `*p = w` we add a χ function for each variable in `p`'s points-to set (i.e., `a`); for the CFG node `u = *v` we add a μ function for each variable in `v`'s points-to set (i.e., `e` and `f`). The other nodes are annotated in a similar fashion. Once all of the nodes have been annotated, any standard SSA algorithm can be used to derive the SSA form.

The def-use information computed by AUX is conservative with respect to the more precise flow-sensitive information that will be computed by the primary analysis, so there are three possibilities that must be addressed for a STORE command `*x = y` that is annotated with $v_m = \chi(v_n)$:

1) `x` might not point to v in the flow-sensitive results. In this case, v_m should be a copy of v_n and incorporate none of `y`'s information.
2) `x` might 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 incorporate none of v_n's information.
3) `x` might point to v as well as other variables in the flow-sensitive results. In this case, the analysis must weakly update the points-to information for v; in other words, v_m should incorporate points-to information from both v_n and `y`.

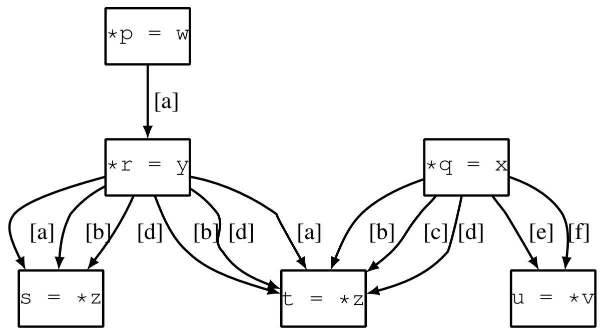


Fig. 4. The def-use graph derived from Figure 3. An edge from statement X to statement Y indicates that a variable potentially defined at X is potentially used at Y. Each edge is labeled with the specific variable that may be defined/used.

By using the computed SSA information to construct the def-use graph, we can accommodate all of these possibilities. To create the DUG we must add an edge from each indirect definition (i.e., STORE) to each statement where the indirectly-defined variable might be used. Thus we create, for each STORE annotated with a function $v_m = \chi(v_n)$, a def-use edge from that STORE to every statement that uses v_m as the argument of a χ, μ, or ϕ function. We label each def-use edge that corresponds to an address-taken variable with the name of that variable; when propagating pointer information, the analysis only propagates a variable's information along edges labeled with that variable's name. Figure 4 shows the program from Figure 2 converted into a DUG based on the SSA information from Figure 3.

In principle, the actual flow-sensitive analysis works much like the traditional flow-sensitive analysis described in Section II, except that (1) the analysis propagates information along def-use edges instead of control-flow edges, and (2) a variable's information is only propagated along edges labeled with that variable's name. The exact steps of the analysis are deferred to Section V after we discuss some important issues that affect precision and performance.

In order for the sparse flow-sensitive analysis to compute a precise solution (i.e., one that is equivalent to a traditional iterative dataflow analysis for flow-sensitive, context-insensitive pointer analysis), it must handle the three STORE possibilities listed above. The key insight required to show 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`. When visiting this node, x must point to something (because 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). The monotonicity property constrains the transitions that the analysis may take among the three possibilities for this STORE: The analysis may transition from possibility (1) or (2) to (3), and from (1) to (2), but it can never transition from possibility (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.

Note that the previous argument assumes that AUX is flow- and context-insensitive; the use of a context-sensitive AUX would instead yield an analysis that is *more precise* than the traditional flow-sensitive, context-insensitive analysis. In particular, a context-sensitive AUX analysis will compute def-use chains that are incomparable to the set of def-use chains that the primary flow-sensitive, context-insensitive analysis would compute (because AUX would be more precise in the context-sensitive dimension than the primary analysis). Thus, with a context-sensitive AUX SFS will compute a more precise (but still sound) result, because the def-use chains used for the sparse primary analysis would benefit from the AUX analysis' context-sensitivity.

C. 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 LOAD and STORE may access thousands of variables, based on the dereferenced variable's points-to set size, and each variable may be accessed at dozens or hundreds of locations in the program—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*, represents 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, the other is too; 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 by Hardekopf and Lin [16]. 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). Location equivalence is not useful in compilers such as LLVM that use partial SSA form; these compilers maintain the invariant that every address-taken variable v has at least one top-level pointer that points only to v—in such a case there are no location equivalent variables, but there can be many access equivalent variables.

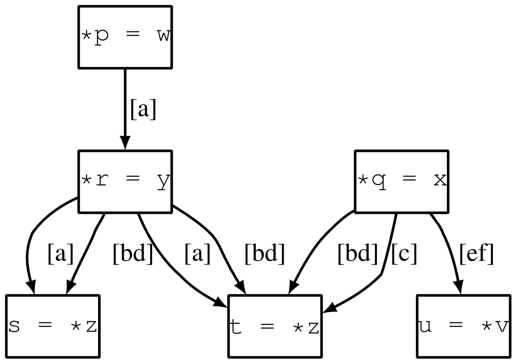


Fig. 5. The def-use graph of Figure 4 after applying access equivalence. Variables with exactly the same set of edges have been grouped together, reducing the total number of edges in the graph.

The advantage of access equivalence is that the SSA algorithm will compute identical def-use chains for all access-equivalent variables. This is easy to see: 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 identify variables that 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.

Once the address-taken variables have been partitioned into access equivalence classes, the edges of the DUG are relabeled using the partitions instead of variable names. For Figure 2, the access equivalences are: $\{a\}$, $\{b, d\}$, $\{c\}$, and $\{e, f\}$. Figure 5 shows the same def-use graph as Figure 4 except with edges for access-equivalent variables in the same partition collapsed into a single edge.

Because the access equivalences are computed using AUX, they are conservative with respect to the actual access equivalences that would be computed using a flow-sensitive pointer analysis—that is, variables that are access equivalent using the AUX results may not be access equivalent using the flow-sensitive results. For this reason, while the DUG edges are labeled using the variable's access equivalence partitions, the transfer functions and points-to graphs at each node use the actual variables and not the partitions.

D. Interprocedural Analysis

There are two possible approaches for extending the above analysis 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 that def-use chains can span a number of functions; treating each function call between the definition and the actual use as a collection point can adversely affect the sparseness of the analysis.

The second option, and the one that we choose, is to compute the sparseness for the entire program as one unit, directly connecting variable definitions and uses even across function boundaries. An important consideration for this approach is the method of handling 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 analysis as given 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 dependences 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 that 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—it has been shown that precise call-graphs can be constructed using only flow-insensitive pointer analysis [25]. 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 dependences. 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.

V. The Final Algorithm

Putting everything together, the final algorithm for sparse flow-sensitive pointer analysis begins with a series of preprocessing steps that computes the DUG:

1) Run AUX to compute conservative def-use information for the program being analyzed. Use the results of AUX to compute the program's interprocedural control-flow graph (ICFG [22]), including the resolution of indirect calls to their potential targets. All function calls are then translated into a set of COPY instructions to represent parameter assignments; similarly, function returns are translated into COPY instructions.
2) Compute exact SSA information for all top-level variables. The ϕ-functions computed by this step are translated into COPY statements, e.g., $x_1 = \phi(x_2, x_3)$ becomes $x_1 = \ x_2\ x_3$ (see **processCopy** below), which distinguishes these statements from the ϕ-functions computed for the address-taken variables in step 4 below. Partition the address-taken variables into access equivalence classes as described in Section IV-C.
3) For each partition P, use the results of AUX to label each STORE that may modify a variable in P with a function $P = \chi(P)$, and label each LOAD that may access a variable in P with a function $\mu(P)$.
4) Compute SSA form for the partitions, using any of many available methods [1], [3], [9], [10].
5) Construct the def-use graph by creating a node for each pointer-related instruction and for each ϕ function created by step 4; then:
   - For each ALLOC, COPY, and LOAD node N, add an unlabeled edge from N to every other node that uses the top-level variable defined by N.
   - 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:

- There is a node worklist $Worklist$, initialized to contain all ALLOC nodes.
- There is 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.
- Every LOAD and ϕ-function k contains a points-to graph IN_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.
- Every STORE node k contains two points-to graphs 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. We lift $\mathcal{P}_k()$ to operate of sets of address-taken variables such that its result is the union of the points-to sets of each variable in the set.
- For each address-taken variable v, $part(v)$ returns the variable partition that v belongs to.

The points-to graphs are all initialized to be empty. The main body of the algorithm is listed below. The loop iteratively selects and processes a node from the worklist, during which new nodes may be added to the worklist. The loop continues until the worklist is empty, at which point the analysis is complete. Each different type of node is processed as given in the code listing below. 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.

```
Main body of analysis:
  while $Worklist$ is not empty do
    $k$ =SELECT($Worklist$)
    switch typeof($k$):
      case ALLOC: processAlloc($k$) // x = ALLOC$_i$
      case COPY:  processCopy($k$)  // x = y z ...
      case LOAD:  processLoad($k$)  // x = *y
      case STORE: processStore($k$) // *x = y
      case $\phi$:     processPhi($k$)   // x = phi(...)
```

```
define processAlloc($k$):
  $PG \hookleftarrow \{x \rightarrow \text{ALLOC}_i\}$
  if $PG$ changed then
    $Worklist \hookleftarrow \{ n \mid k \rightarrow n \in E\}$
```

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

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

```
define processStore($k$):
  if $\mathcal{P}_{top}(x)$ represents a single concrete memory location then
    // strong update
    $\text{OUT}_k \hookleftarrow (\text{IN}_k - \{\mathcal{P}_{top}(x) \rightarrow \_\}) \cup \{\mathcal{P}_{top}(x) \rightarrow \mathcal{P}_{top}(y)\}$
  else // weak update
    $\text{OUT}_k \hookleftarrow \text{IN}_k \cup \{\mathcal{P}_{top}(x) \rightarrow \mathcal{P}_{top}(y)\}$
  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 $\text{IN}_n$ changed then
        $Worklist \hookleftarrow \{n\}$
```

```
define processPhi($k$):
  for all $\{n \in N \mid k \rightarrow n \in E\}$ do
    $IN_n \hookleftarrow IN_k$
    if $\text{IN}_n$ changed then
      $Worklist \hookleftarrow \{n\}$
```

Name	Description	LOC
197.parser	parser	11K
300.twolf	place and route simulator	20K
ex	text processor	34K
255.vortex	object-oriented database	67K
254.gap	group theory interpreter	71K
sendmail	email server	74K
253.perlbmk	PERL language	82K
nethack	text-based game	167K
python	interpreter	185K
176.gcc	C language compiler	222K
vim	text processor	268K
pine	e-mail client	342K
svn	source control	344K
ghostscript	postscript viewer	354K
gimp	image manipulation	877K
tshark	wireless network analyzer	1,946K

TABLE I
BENCHMARKS: **LOC** REPORTS THE NUMBER OF LINES OF CODE. THE BENCHMARKS ARE DIVIDED INTO SMALL (LESS THAN 100K LOC), MID-SIZED (BETWEEN 100K–400K LOC), AND LARGE (GREATER THAN 800K LOC).

VI. EVALUATION

To evaluate the scalability of our new technique, we compare it against the most scalable known flow-sensitive pointer analysis algorithm—semi-sparse flow-sensitive pointer analysis (SSO) [17]. SSO is able to analyze benchmarks with up to approximately 344K lines of code (LOC), an order of magnitude greater than (and almost 200× faster than) allowed by the best traditional, non-sparse flow-sensitive pointer analysis. 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 and uses the method indicated in Section IV-D for handling function pointers. All of these algorithms—SSO, SFS, and AUX–are field-sensitive, meaning that each field of a struct is treated as a separate variable.

Both SFS and SSO are implemented in the LLVM compiler infrastructure [23] and use BDDs from the BuDDy library [24] to store points-to relations. We emphasize that neither technique is a symbolic analysis (which formulates the entire analysis as operations on relations represented as boolean functions). Instead, SFS and SSO only use BDDs to compactly represent points-to sets; other data structures could be swapped in for this purpose without changing the rest of the analysis. The analyses are written in C++ and handle all aspects of the C language except for varargs.

The benchmarks for our experiments are described in Table I. Six of the benchmarks are the largest SPECint 2000 benchmarks, and the rest are various open-source applications. 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.

A. Performance Results

Table II 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 solver stage.

The premise of SFS—that a sparse analysis can be approximated by using an auxiliary pointer analysis to conservatively compute def-use information—is clearly borne out. For the small benchmarks, those less than 100K LOC, both analyses are fast, but the advantage is unclear: In some cases SFS is faster, and in other cases SFS is slower; on average SFS is 1.03× faster than SSO. For the mid-sized benchmarks, those with between 100K LOC and 400K LOC, SFS has a more distinct advantage; for the six benchmarks that both algorithms complete, SFS is on average 5.5× faster than than SSO. The scalability advantage of SFS is seen from the fact that SSO cannot analyze the three largest benchmarks within an hour.

The one area where SSO has a clear advantage is memory consumption, but SFS has not been tuned with respect to memory consumption, and we believe its memory footprint can be significantly reduced.

B. 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, but remember that the AUX column includes the time for AUX to generate constraints, optimize those constraints, solve them, and 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.

Second, we 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.

VII. Conclusions

The ability to perform a sparse analysis is critical to the scalability of any flow-sensitive analysis. In this paper, we have shown how pointer analysis can be performed sparsely with a staged approach. In particular, our algorithm uses a highly efficient inclusion-based pointer analysis to create conservative def-use information, and from this information the algorithm performs a sparse flow-sensitive pointer analysis. Our new algorithm is quite scalable even though it has not yet been carefully tuned. In particular, we have identified a number of memory optimizations that should reduce its high memory requirements, and other optimizations should improve its already fast analysis time.

This paper presents the first study of staged pointer analysis, a family of algorithms that we believe deserves further exploration. In particular, the idea of a context-sensitive auxiliary analysis is an intriguing idea for future work. As we have mentioned, a combined flow- and context-sensitive analysis limits scalability, but by obtaining context-sensitive and flow-sensitive information in sequence rather than in a combined analysis, this new staged analysis promises to offer significantly greater scalability than previous flow- and context-sensitive algorithms, albeit with less precision.

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

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

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

[5] 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.

[6] 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.

[7] 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.

[8] 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 (CC)*, 1996.

[9] 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.

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

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

[12] 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.

[13] R. Ghiya. Putting pointer analysis to work. In *Principles of Programming Languages (POPL)*, 1998.

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

Name	SSO		SFS					Time	Mem
	Time	Mem	Prelim	Prep	Solve	Total Time	Mem	Comp	Comp
197.parser	0.41	138	0.29	0.07	0.008	0.37	275	1.11	0.50
300.twolf	0.23	140	0.34	0.07	0.004	0.41	281	0.56	0.50
ex	0.35	141	0.29	0.10	0.008	0.40	277	0.88	0.51
255.vortex	0.60	144	0.45	0.14	0.028	0.62	285	0.97	0.51
254.gap	1.28	155	0.94	0.33	0.016	1.29	307	0.99	0.50
sendmail	1.21	147	0.70	0.27	0.032	1.00	301	1.21	0.49
253.perlbmk	2.30	158	1.05	0.50	0.020	1.57	312	1.46	0.51
nethack	3.16	197	1.72	0.82	0.096	2.64	349	1.20	0.56
python	120.16	346	4.04	2.02	0.564	6.62	404	18.15	0.86
175.gcc	3.74	189	2.00	1.42	0.040	3.46	370	1.08	0.51
vim	61.85	238	2.93	2.44	0.160	5.53	436	11.18	0.55
pine	347.53	640	13.42	21.25	47.330	82.00	876	4.24	0.73
svn	185.10	233	5.40	5.07	0.216	10.69	418	17.32	0.56
ghostscript	OOT	—	42.98	86.13	1787.184	1916.29	2359	∞	0
gimp	OOT	—	90.59	105.87	1025.824	1222.28	3273	∞	0
tshark	OOT	—	232.54	219.83	376.096	828.47	6378	∞	0

TABLE II

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. TIME COMP IS THE SSO TIME DIVIDED BY THE SFS TIME (I.E., SPEEDUP; LARGER IS BETTER); MEM COMP IS THE SSO MEMORY DIVIDED BY THE SFS MEMORY (LARGER IS BETTER).

[15] 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.

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

[17] B. Hardekopf and C. Lin. Semi-sparse flow-sensitive pointer analysis. In *Symposium on Principles of Programming Languages (POPL)*, 2009.

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

[19] 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.

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

[21] V. Kahlon. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In *Programming Language Design and Implementation (PLDI)*, pages 249–259, 2008.

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

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

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

[25] A. Milanova, A. Rountev, and B. 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.

[26] D. Novillo. Design and implementation of Tree SSA. In *GCC Developers Summit*, pages 119–130, 2004.

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

[28] E. Ruf. Partitioning dataflow analyses using types. In *Symposium on Principles of Programming Languages (POPL)*, pages 15–26, 1997.

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

[30] B. Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages (POPL)*, pages 32–41, New York, NY, USA, 1996. ACM Press.

[31] 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.

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

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

[34] 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.