Analyzing Millions of Lines of Code with Sparse Flow-Sensitive Pointer Analysis

Ben Hardekopf Calvin Lin

The University of Texas at Austin

{benh,lin}@cs.utexas.edu

Abstract

Many program analyses perform better and produce more precise results when they are given precise pointer information. One dimension of precision is flow-sensitive pointer analysis (i.e., an analysis that respects a program's control-flow), which has been shown to be useful for important applications such as program verification and understanding, among others. However, this type of pointer analysis has historically been unable to scale to programs with hundreds of thousands of lines of code.

We present a new interprocedural flow-sensitive pointer analysis that has significantly better performance than the current state-of-the-art. There are two key ideas behind our algorithm: (1) We *stage* the pointer analysis by first running a flow-insensitive auxiliary pointer analysis that produces conservative def-use information; the primary flow-sensitive analysis can then employ a sparse analysis that efficiently propagates pointer information across these conservative def-use chains; and (2) we exploit a novel notion of equivalence called *access equivalence*, which allows the algorithm to efficiently apply the conservative def-use information for the sparse analysis even when the conservative nature of the auxiliary analysis produces a huge amount of def-use information. Together, these techniques produce the first flow-sensitive pointer analysis that can analyze a C program with 1.9M lines of code, an order of magnitude improvement over the previous state-of-the-art.

1. Introduction

The performance and precision of many program analyses can be improved by supplying the analyses with precise pointer information. While many dimensions of pointer analysis precision exist, the poor scalability of the more precise analyses typically forces practitioners to forsake precision in favor of performance. Recently, however, substantial progress has been made in various dimensions of precision. Context-sensitive pointer analysis is significantly more efficient with the advent of BDD-based symbolic analysis [2, 30, 31, 32, 33]. Similarly, the performance of inclusion-based analysis, which is the most precise of the flow-insensitive analyses, has been significantly improved in terms of both time and memory [14, 15]. However, flow-sensitive pointer analysis has not seen as much progress, even though it has been shown to be useful for a number of program analyses [6, 12], including those that check for security vulnerabilities [4, 11, 13], those that synthesize hardware [32], and those that analyze multi-threaded codes [27]. Flow-sensitive pointer analysis has historically been unable to analyze programs with more than a few tens of thousands of lines of code.

Flow sensitive pointer analysis, which traditionally uses an iterative data-flow analysis framework, is notoriously inefficient. Because the analysis does not know where pointer information is required, the analysis needlessly pushes pointer information to every reachable node in the program, even those that cannot use those data-flow facts. For large benchmarks, the control flow graph (CFG) can have hundreds of thousands of nodes; each node maintains two points-to graphs, one for incoming information and one for outgoing information; each points-to graph in turn can have hundreds of thousands of pointers; and each pointer can have thousands of elements in its points-to set. Thus, each node stores and propagates an enormous amount of useless information.

The typical method of optimizing a flow-sensitive data-flow analysis is to perform a *sparse analysis* [5, 26], 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. Thus, sparse analysis greatly improves the efficiency of the algorithm in terms of both time and memory.

Unfortunately, pointer analysis does not lend itself to a sparse analysis because pointer information is required to compute the very def-use chains that enable a sparse analysis. This paper describes a new flow-sensitive pointer analysis algorithm that overcomes this challenge.

1.1 Insights

The key insight behind our algorithm is to *stage* the analysis using a flow-insensitive *auxiliary* pointer analysis to compute conservative def-use information. Armed with this conservative def-use information, the primary flow-sensitive analysis can then be performed sparsely. For this idea to work, the balance between the precision and performance of the auxiliary analysis is critical. If the results are too imprecise, the sparsity of the primary analysis will suffer. If the auxiliary analysis is not scalable, the primary analysis will never get to execute. With its recent advances [14, 15], we believe that inclusion-based analysis could lie in a sweet spot, because it can analyze millions of lines of code in a matter of minutes and because, as mentioned earlier, it is the most precise of the flow-insensitive pointer analyses.

The primary difficulty of staged analysis is the complexity of incorporating flow-insensitive def-use information while maintaining the precision of a flow-sensitive pointer analysis. A second difficulty is the sheer number of def-use chains that are computed by the auxiliary analysis. Each store through a pointer creates a new definition for each variable in that pointer's points-to set. Each load through a pointer creates a new use for each variable in that pointer's points-to set. For large benchmarks, each pointer can have thousands of elements in its points-to set, so the auxiliary analysis can create hundreds of millions of def-use chains. We address both of these issues in Section 4.

1.2 Contributions

This paper makes the following contributions:

- We introduce a new flow-sensitive pointer analysis, called *staged flow-sensitive analysis,* which 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 the conservative def-use chains into equivalence classes, allowing our algorithm to efficiently process the huge amount of def-use information.
- Using a collection of 16 open source C programs, we evaluate our new algorithm by comparing it to a baseline which represents the prior state-of-the-art [16]. Our staged algorithm can analyze a program with 1.9M LOC in under 14 minutes. This program size is an order of magnitude larger than the baseline's largest program (344K LOC), and the baseline algorithm itself represents an order of magnitude improvement over its preceding state-of-the-art. In comparing our new algorithm with the baseline, we see that for small programs—those with fewer than 100K LOC, our new algorithm provides no performance advantage. For mid-sized programs—those with 100K to 400K LOC, our new algorithm is 5.5× faster. And for large programs—those with more than 800K LOC, our algorithm completes while the baseline does not.

The remainder of this paper is organized as follows. Section 2 provides important background information for understanding this paper. Section 3 contrasts our work with prior research. Sections 4 and 5 then describe our new algorithm. Our experimental evaluation is given in Section 6, and we conclude in Section 7.

2. Background

This section provides background that is necessary for understanding the remainder of the paper. We first describe flow-sensitive pointer analysis, then we describe static single assignment form, and then we describe the particular compiler infrastructure that we use for this work.

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

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

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

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

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 SSA

Static single assignment (SSA) form is a common intermediate representation that requires all variables to be defined exactly once in the text of the program. Variables defined multiple times in the original representation are split into separate instances, one for each definition. When separate instances of the same variable are live at a join point in the control-flow graph, they are combined using a ϕ function, which takes the old instances as arguments and assigns the result to a new instance. SSA form is ideal for performing sparse analyses because it makes def-use information explicit in the program representation and allows data-flow information to flow directly from variable definitions to their corresponding uses [26].

There are many known algorithms for converting a program into SSA form [1, 3, 8, 9]. However, the problem becomes more difficult when we consider indirect definitions and uses through pointers. These definitions and uses can only be discovered using pointer analysis. Because of the conservative nature of the pointer analysis results, each indirect definition and use is actually a *possible* definition or use. Following Chow et al [7], we use χ and μ functions to represent these possible definitions and uses. Each STORE in the original program representation (i.e., prior to the transformation into SSA form) is annotated with a function $v = \chi(v)$ for each variable v that may be defined by the STORE; similarly, each LOAD in the original representation is annotated with a function $\mu(v)$ for each variable v that may be accessed by the LOAD. When converting to SSA form, each χ function is treated as both a definition and use of the given variable, and each μ function is treated as a use of the given variable. The χ function represents the fact that a variable may not be defined at the associated STORE and therefore copies the old value of the variable into the new instance. The way to interpret a STORE with an associated χ function for variable v is that the STORE may define v (in which case its value is the right-hand side of the STORE) or it may not (in which case its value is unchanged).

To avoid these problems, modern compilers such as GCC [25] and LLVM [22] use a variant of SSA, which we call *partial* SSA

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

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

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

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

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

form. The key idea is to divide variables into two classes. One class contains variables that are never referenced by pointers (*top-level variables*), so their definitions and uses can be trivially determined by inspection without pointer information, and these variables can be converted to SSA using any algorithm for constructing SSA form. The other class contains those variables that *can* be referenced by pointers (*address-taken variables*), and these variables are not placed in SSA form because of the above-mentioned complications.

2.3 The LLVM IR

We use the LLVM compiler infrastructure to implement our algorithm, so we now describe 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. 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, and they are not in SSA form. Top-level variables are modified using ALLOC 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.

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.

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.

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.

3. 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 [29], whose algorithm could analyze C programs with almost 70,000 lines of code in roughly 30 minutes. The scalability is limited by the cost of dynamically updating SSA form.

Our previous work [16] uses SSA form where possible and uses iterative data-flow analysis elsewhere. More precisely, SSA form is used for top-level variables, which by definition are not complicated by pointer accesses. Thus, the uses and definitions of top-level pointers can be trivially determined by inspection, and these variables can be converted to SSA without performing any pointer analysis. By contrast, our staged analysis uses SSA form for all variables.

Hasti and Horwitz [17] 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, our staged algorithm uses a single pass of the the flow-insensitive analysis to compute a fully precise flow-sensitive analysis.

Hind and Pioli [18, 19] use a weaker form of sparsity in the form of the *sparse evaluation graph* (SEG), which is a graph that is derived from the CFG by eliding nodes that are irrelevant to pointer analysis because they do not manipulate pointer information, while at the same time maintaining the control-flow relations among the remaining nodes. Hind and Pioli [18, 19] 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 or from one of the function parameters. Hind and Pioli's algorithm scales to about 30K LOC.

Previous work has also used notions similar to staging to improve the efficiency of pointer analysis. Client-driven pointer analysis [13] 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. Fink et al [11] apply 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. Kahlon bootstraps the flow-sensitive pointer analysis by using a flow-insensitive pointer analysis to first partition the program into sections that can be analyzed independently [20]. Unlike staging, these techniques use a preliminary analysis to reduce the size of the input by either pruning [11, 13] or partitioning the program [20]. 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.

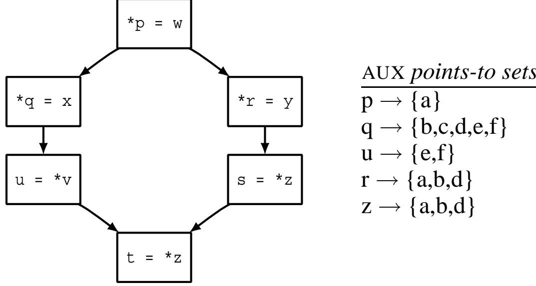

Figure 3. Example CFG, along with a subset of the points-to sets computed by AUX.

4. Staging the Analysis

The essential idea of our new algorithm is to enable sparse flow-sensitive pointer analysis for *all* variables in the program by staging the pointer analysis. We first employ an auxiliary, flow-insensitive pointer analysis to compute conservative def-use information for a program; we then use that information to increase the sparseness of the primary, flow-sensitive pointer analysis, thereby greatly increasing its efficiency. The remainder of this section discusses the auxiliary pointer analysis and how to use its results to optimize the primary analysis.

4.1 Auxiliary Pointer Analysis

Any flow-insensitive pointer analysis can be used for the auxiliary analysis. There are many to choose from, ranging from the simplest address-taken analysis (which reports that any pointer can point to any variable whose address has been taken), to Steensgaard's analysis [28], to Das' One-Level Flow [10], to inclusion-based (i.e., Andersen-style) analysis, which is the most precise of all these analyses. In choosing an auxiliary pointer analysis, there are two important considerations: (1) how scalable the auxiliary analysis is, and (2) how effective its results are for optimizing the primary, flow-sensitive analysis. The more precise the auxiliary analysis, the more sparse the primary analysis will be; for this reason, and because recent work has made inclusion-based analysis extremely scalable [14, 15], we believe that inclusion-based analysis is the best choice. Henceforth, we will designate the chosen auxiliary pointer analysis as AUX.

4.2 Sparse Flow-Sensitive Pointer Analysis

The primary data structure that we use for the sparse flow-sensitive pointer analysis is a def-use graph (DUG). The DUG contains a node for each statement in the program, and its edges represent def-use chains—if a variable is defined in node x and used in node y, there is a directed edge from x to y. The def-use edges for top-level variables are trivial to determine from inspection of the program; the def-use edges for address-taken variables require AUX to compute. This section describes how 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. We annotate the LOADs and STOREs using χ and μ functions as described in Section 2.2, then convert the program to SSA form using any standard SSA algorithm [1, 3, 8, 9]. Figure 3 shows a small example program along with some pointer information discovered by AUX. Figure 4 shows the same example program, annotated with χ and μ functions and translated into SSA.

Note that the def-use information revealed by the χ and μ functions and SSA form is conservative with respect to the more precise flow-sensitive information that will be computed by the primary

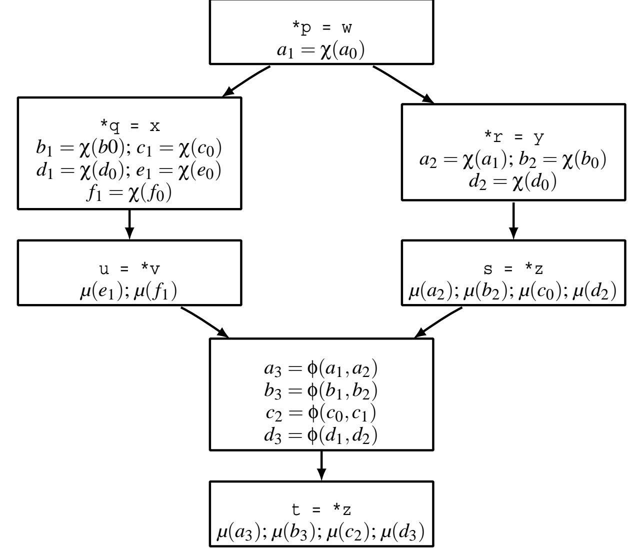

Figure 4. The SSA information for Figure 3.

analysis. In particular, there are three possibilities that must be addressed for a STORE `*x = y` that is annotated with $v_m = \chi(v_n)$:

1. `x` may not point to v in the flow-sensitive results. In this case, the analysis should interpret the STORE based solely on the χ function; in other words, v_m should be a copy of v_n and not incorporate `y` at all.
2. `x` may point only to v in the flow-sensitive results. In this case, the analysis can strongly update the points-to information for v; in other words, v_m should be a copy of `y` and not incorporate v_n at all.
3. `x` may point to v as well as other variables in the flow-sensitive results. In this case, the analysis must weakly update the points-to information for v; in other words, v_m should incorporate the points-to information from both v_n and `y`.

By using the SSA form to fill in the def-use graph DUG, we can accommodate all of these possibilities. For each STORE annotated with a function $v_m = \chi(v_n)$, we create a def-use edge to every statement that uses v_m as the argument of a χ, μ, or ϕ function. We label each def-use edge for an address-taken variable with that variable, so the analysis can determine along which edge to propagate a given variable's points-to information. Figure 5 shows the example program from Figure 3 converted into a def-use graph based on the SSA information from Figure 4.

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

Theorem 1 (The Analysis is Correct)**.** *Every definition of a variable reaches its corresponding uses, and the analysis computes precise flow-sensitive pointer information.*

Proof. We prove the theorem in two parts:

Every def reaches its corresponding uses. Points-to information flows along the def-use chains in DUG computed by AUX. Since

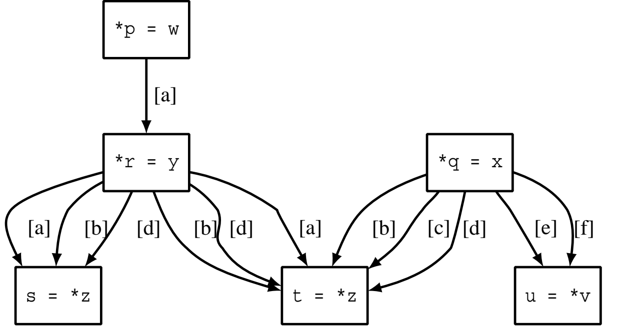

Figure 5. The def-use graph for Figures 3 and 4; each edge is labeled with the variables used by the destination.

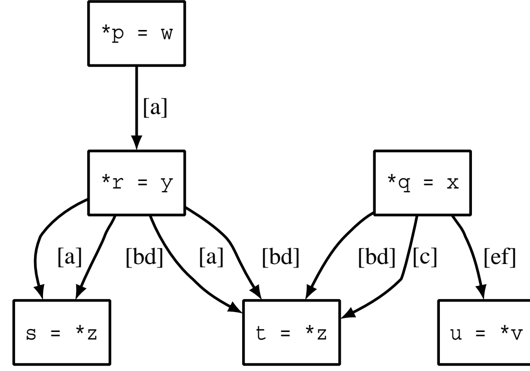

Figure 6. The def-use graph of Figure 5 after applying access equivalence.

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

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

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

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

4.3 Access Equivalence

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

Two address-taken variables x and y are access equivalent if whenever one is accessed by a LOAD or STORE instruction, so is the other; in other words, for all variables v such that v is dereferenced in a LOAD or STORE, $x \in points\text{-}to(v) \Leftrightarrow y \in points\text{-}to(v)$. This notion of equivalence is similar, but not identical to the notion of *location equivalence* described by Hardekopf and Lin [15]. 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).

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

For Figure 3, the access equivalences are: $\{a\}, \{b,d\}, \{c\}, \{e,f\}$. Figure 6 shows the same def-use graph as Figure 5 except with edges for access-equivalent variables collapsed into a single edge.

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

4.4 Interprocedural Analysis

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

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

There are two possible solutions to this problem. The easiest is simply to assume the AUX computes a precise call-graph, i.e., the same call-graph the flow-sensitive pointer analysis would compute. If AUX is fairly precise (e.g., an inclusion-based analysis), this is a good assumption to make—it has been shown that precise call-graphs can be constructed using only flow-insensitive pointer analysis [24]. We use an inclusion-based analysis for AUX, and hence this is the solution we use for our work.

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

5. The Final Algorithm

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

1. Run AUX to compute conservative def-use information for the program being analyzed.
2. Use the results of AUX to compute the interprocedural control-flow graph (ICFG [21]) of the program, including resolving indirect calls to their potential targets. All function calls are then translated into a set of COPY instructions to represent parameter assignments, and similarly function returns are also translated into COPY instructions.
3. Compute exact SSA information for all top-level variables.
4. Partition the address-taken variables into access equivalence classes as described in Section 4.3.
5. For each partition P, use the results of AUX to:
   - Label each STORE that may modify a variable in P with a function $P = \chi(P)$.
   - Label each LOAD that may access a variable in P with a function $\mu(P)$.
6. Compute SSA form for the partitions, using any of many available methods (e.g., [1, 3, 8, 9]).
7. Construct the def-use graph by creating a node for each pointer-related instruction and each ϕ function created by step 6, then:
   - For each ALLOC, COPY, and LOAD node N, add an unlabeled edge from N to every other node that uses the variable defined by N (note that because of step 3, nodes of these types each define a unique variable; the COPY nodes include the ϕ functions computed by step 3).
   - For each STORE node N that has a χ function defining a partition variable P_n, add an edge from N to every node that uses P_n (either in a ϕ, χ or μ function), labeled by the partition P.
   - For each ϕ node N that defines a partition variable P_n, create an unlabeled edge to every node that uses P_n.

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

- 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 node 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.
- For each address-taken variable v, $part(v)$ returns the variable partition that v belongs to.

The main body of the algorithm is listed in Algorithm 1. The loop iteratively selects a node from the worklist and processes it, which may add new nodes to the worklist. It continues until the worklist is empty, at which point the analysis is complete. Each different type of node is processed as listed in Algorithms 2–6. The $\hookleftarrow$ operator represents set update, $\rightarrow$ represents an unlabeled edge in the def-use graph, and $\xrightarrow{x}$ represents an edge labeled with x.

Algorithm 1 Main body of the semi flow-sensitive pointer analysis algorithm.

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

Algorithm 2 processAlloc(k) : [x = ALLOC$_i$]

```
PG ↩ {x → ALLOC_i}
if PG changed then
  Worklist ↩ { n | k → n ∈ E }
```

Algorithm 3 processCopy(k) : [x = y z ...]

```
for all v ∈ right-hand side do
  PG ↩ {x → P_top(v)}
if PG changed then
  Worklist ↩ { n | k → n ∈ E }
```

Algorithm 4 processLoad(k) : [`x = *y`]

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

Algorithm 5 processStore(k) : [`*x = y`]

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

Algorithm 6 processPhi(k)

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

5.1 Further Optimization

In addition to the techniques described earlier, we use two optimizations described in our earlier work [16]. We briefly recap each optimization here.

Top-Level Pointer Equivalence. This optimization identifies variables that are *pointer equivalent*, i.e., they are guaranteed to have identical points-to sets. Pointer-equivalent variables can be collapsed to reduce the input size of the pointer analysis.

Local Points-to Graph Equivalence. This optimization identifies LOAD and STORE nodes in the def-use graph that are guaranteed to have identical points-to graphs. These nodes can share a single points-to graph among themselves, which saves space and avoids unnecessary propagation of pointer information between nodes.

6. Evaluation

To evaluate our new technique, we compare it against our earlier work on flow-sensitive pointer analysis, called SSO, which is the most scalable algorithm available [16]. SSO is able to analyze benchmarks with up to approximately 344K lines of code (LOC), an order of magnitude greater than allowed by the previous state of the art, and it is almost 200× faster than the previous state of the art. We use SSO as the baseline for comparison with our new technique, which we refer to as SFS. SFS uses inclusion-based (i.e., Andersen-style) analysis for AUX. SSO, SFS, and AUX are all field-sensitive—each field of a struct is treated as a separate variable.

Both SSO and SFS are implemented in the LLVM compiler infrastructure [22] and use BDDs to store points-to relations. We emphasize that neither technique is a symbolic analysis (such as the various symbolic pointer analysis that have been described in other work [2, 30, 31, 32, 33])—instead, we only use BDDs to compactly represent points-to sets; we could swap in other data structures for this purpose without changing the rest of the analysis. We make use of the BuDDy BDD library [23]. The analyses 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 1. Six of the benchmarks are taken from SPECINT 2000 (the largest six applications from that suite) and the rest from 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.

6.1 Performance Results

Table 2 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 approximating a sparse analysis by using an auxiliary pointer analysis to conservatively compute def-use information—is clearly borne out. For the smaller benchmarks, those less than 100K LOC, the advantage is less clear; sometimes SSO is faster, sometimes SFS is faster. For the benchmarks with less than 100K LOC, SFS is on average 1.03× faster than SSO. For the mid-sized benchmarks, those with between 100K LOC and 400K LOC, SFS has a more distinct advantage; it is on average 5.5× faster than than SSO for the six benchmarks that both algorithms complete. In addition, SFS successfully analyzes three benchmarks, each in less than $\frac{1}{2}$ hour, that SSO cannot analyze within an hour.

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

6.2 Performance Discussion

There are several observations about the SFS results that may seem surprising.

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

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

Finally, the prep time for SFS, which includes the time to compute SSA information using the AUX results and the time to optimize the analysis using Top-level Pointer Equivalence and Local Points-to Graph Equivalence, takes a significant portion of the total time for SFS. While the prep stage is compute-intensive, there

Name	Description	LOC	Statements	TL Vars	AT Vars
197.parser	parser	11K	18K	7.6K	1.9K
300.twolf	place and route simulator	20K	37K	12.4K	4.8K
ex	text processor	34K	37K	8.8K	2.3K
255.vortex	object-oriented database	67K	47K	15.3K	5.9K
254.gap	group theory interpreter	71K	99K	39.8K	8.2K
sendmail	email server	74K	54K	20.2K	28.5K
253.perlbmk	PERL language	82K	118K	48.8K	4.1K
nethack	text-based game	167K	298K	79.0K	15.1K
python	interpreter	185K	162K	70.7K	21.9K
176.gcc	C language compiler	222K	258K	108.0K	12.9K
vim	text processor	268K	249K	74.8K	168.0K
pine	e-mail client	342K	426K	206.0K	404.0K
svn	source control	344K	158K	83.5K	23.8K
ghostscript	postscript viewer	354K	388K	164.0K	350.0K
gimp	image manipulation	877K	929K	408.0K	146.0K
tshark	wireless network analyzer	1,946K	1,045K	914.0K	641.0K

Table 1. Benchmarks: **LOC** reports the number of lines of code, **Statements** reports the number of statements in the LLVM IR, **TL Vars** reports the number of top-level variables, and **AT Vars** reports the number of address-taken variables. The benchmarks are divided into small (less than 100K LOC), mid-sized (between 100K–400K LOC), and large (800K LOC and greater).

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

Table 2. 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.

are several optimizations for this stage that we have not yet implemented. We believe that the times for this stage can be significantly reduced.

To better understand the results, we focus on three key aspects of SFS that contribute to its success, the analysis' sparsity, the effect of access equivalence, and the effects of local points-to graph equivalence [16].

The first aspect is the effect of using a sparse analysis for the address-taken variables. We measure this effect by counting, for each address-taken variable v, the number of edges that v's points-to information can propagate across. The sparser the analysis, the fewer edges a variable's information will propagate across, and the more quickly the analysis will complete. Figure 7 shows a comparison, for each benchmark, of the average number of edges a variable's information will propagate across for a non-sparse analysis versus SFS's sparse analysis. As expected, the sparse analysis propagates information across far fewer edges for every benchmark.

The second aspect is the effect of exploiting access equivalence. We use access equivalence to partition address-taken variables so that we only need def-use edges per partition, rather than per variable. Figure 8 compares the number of partitions versus the number of address-taken variables, and also the number of def-use edges used for partitions versus the number of edges that would be required if they were per-variable. Most of the benchmarks, and all of the larger benchmarks, show a significant reduction in the number of edges required. For the larger benchmarks, this reduction in absolute terms was from hundreds of millions of edges to millions of edges.

The final aspect that we consider is the effect of the local points-to graph equivalence optimization [16] that was used for both SSO and SFS. Figure 9 shows the percentage of the number of nodes that remain after merging nodes that share local points-to graphs. We see that the optimization is quite effective.

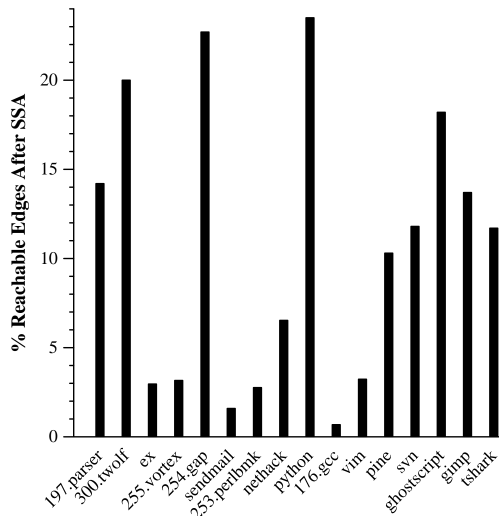

Figure 7. This graph reveals the sparsity of the def-use graph by giving the percentage of edges across which pointer information will be propagated in the def-use graph in relation to a non-sparse analysis using the CFG. Smaller is better: the smaller the percentage, the fewer edges a variable's pointer information will be propagated across.

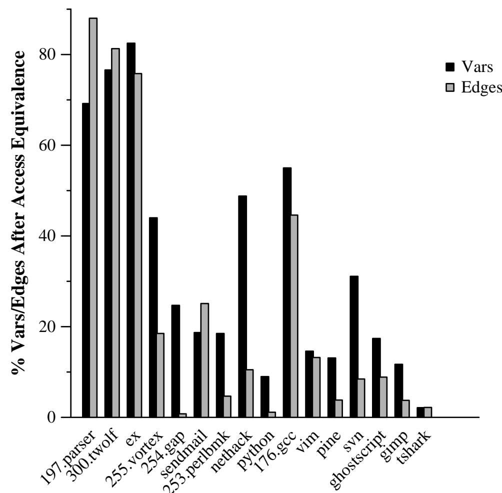

Figure 8. This graph shows the effectiveness of the access equivalence optimization in two ways: Vars is the remaining number of variables after replacing each variable with a representative from its access equivalence class; Edges is the number of remaining def-use edges after merging edges for variables from the same access equivalence class. Both are given as a percentage of the number of variables and def-use edges without using access equivalence. Smaller is better: the smaller the percentage, the fewer variables and edges remain in the graph.

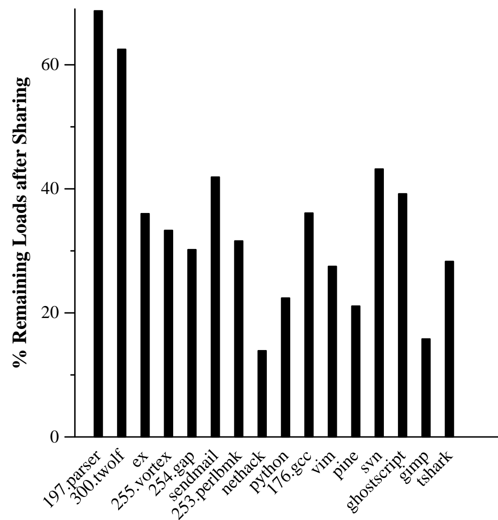

Figure 9. This graph shows the effectiveness of sharing points-to graphs using local points-to graph equivalence by giving the number of LOAD instructions that remain after merging nodes that share points-to graphs, as a percentage of the total number of LOADs. Smaller is better: the smaller the percentage of remaining nodes, the more sharing is being done.

7. Conclusion

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. While our new algorithm is quite scalable, 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 represents just the first study of staged pointer analysis. It would be interesting to see how a less precise auxiliary analysis would perform, and it would be interesting to apply the broader idea of staging—use a less precise analysis to optimize a more precise analysis—to other pointer analyses.

References

[1] John Aycock and R. Nigel 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] Marc Berndl, Ondrej Lhotak, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using BDDs. In *Programming Language Design and Implementation (PLDI)*, 2003.

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

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

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

[6] Peng-Sheng Chen, Ming-Yu Hung, Yuan-Shin Hwang, Roy Dz-Ching Ju, and Jenq Kuen Lee. Compiler support for speculative multithreading architecture with probabilistic points-to analysis. *SIGPLAN Not.*, 38(10):25–36, 2003.

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

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

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

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

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

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

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

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

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

[16] Ben Hardekopf and Calvin Lin. Semi-sparse flow-sensitive pointer analysis. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, 2009 (to appear).

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

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

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

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

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

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

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

[24] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Precise and efficient call graph construction for c programs with function pointers. *Automated Software Engineering special issue on Source Code Analysis and Manipulation*, 11(1):7–26, 2004.

[25] Diego Novillo. Design and implementation of tree ssa, 2004.

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

[27] Alexandru Salcianu and Martin 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.

[28] Bjarne 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.

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

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

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

[32] Jianwen 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.

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