Scalable Flow-Sensitive Pointer Analysis for Java with Strong Updates

Arnab De and Deepak D'Souza

Department of Computer Science and Automation,
Indian Institute of Science, Bangalore, India
{arnabde,deepakd}@csa.iisc.ernet.in

Abstract. The ability to perform strong updates is the main contributor to the precision of flow-sensitive pointer analysis algorithms. Traditional flow-sensitive pointer analyses cannot strongly update pointers residing in the heap. This is a severe restriction for Java programs. In this paper, we propose a new flow-sensitive pointer analysis algorithm for Java that can perform strong updates on heap-based pointers effectively. Instead of points-to graphs, we represent our points-to information as maps from access paths to sets of abstract objects. We have implemented our analysis and run it on several large Java benchmarks. The results show considerable improvement in precision over the points-to graph based flow-insensitive and flow-sensitive analyses, with reasonable running time.

1 Introduction

Pointer analysis is used to determine if a pointer may point to an abstract memory location, typically represented by an allocation site in languages like Java. A precise pointer analysis has the potential to increase the precision and scalability of client program analyses [29,17]. The precision of pointer analysis can be improved along two major dimensions: *flow-sensitivity* and *context-sensitivity*. A flow-insensitive pointer analysis [1,31] computes a single points-to information for the entire program that over-approximates the possible points-to relations at all states that the program may reach at run-time. A flow-sensitive analysis on the other hand takes the control flow structure of a program into account and produces separate points-to information at every program statement. A context-sensitive analysis aims to distinguish among invocations of the same function based on the calling contexts.

Traditionally researchers have focused on improving the scalability and precision of flow-insensitive [14,2,26,10] and context-sensitive analyses [25,34,33]. Flow-sensitive analyses were found to be expensive and gave little additional payoff in client applications like memory access optimizations in compilers [16,15,17]. However in recent years, it has been observed that several client analyses like typestate verification [8], security analysis [5], bug detection [9], and the analysis of multi-threaded programs [28], can benefit from a precise flow-sensitive pointer

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 665–687, 2012.

analysis. As a result there has been renewed interest in the area of flow-sensitive pointer analysis and the scalability of such analyses, particularly for C programs, has been greatly improved [12,22,38,21,11,18].

Most of these techniques however compute the points-to information as a *points-to graph* (or some variant of it), which as we explain below, can be a severe limitation to improvements in precision for Java programs. A node in a points-to graph can be a variable or an abstract object representing a set of dynamically allocated heap objects. Figure 1(b) shows an example points-to graph. Typically, allocation sites are used as abstract objects to represent all concrete objects allocated at that site. An edge from a variable to an abstract object denotes that the variable may point to that object. Similarly an edge from an abstract object `o1` to an abstract object `o2`, annotated with field `f`, denotes that the `f` field of object `o1` may point to the object `o2`[1]. Precision improvements of flow-sensitive pointer analyses come mostly from the ability to perform *strong updates* [21]. If the analysis can determine that an assignment statement writes to a single concrete memory location, it can *kill* the prior points-to edges of the corresponding abstract memory location. It requires the lhs of the assignment to represent a single abstract memory location *and* that abstract memory location to represent a single concrete memory location. As abstract objects generally represent multiple concrete objects, the analysis cannot perform a strong update on such objects. This situation is common in Java programs, where all indirect assignment statements (i.e. assignments whose lhs have at least one dereference) write to the heap, and hence traditional flow-sensitive algorithms cannot perform any strong updates for such assignments.

We illustrate this problem using the program fragment of Figure 1(a). The points-to graph before statement `L1` is shown in Figure 1(b), where variables `p` and `r` point to the abstract heap location `o1`, `q` points to `o3`, and field `f` of object `o1` points to the object `o2`. If the abstract object `o1` represents multiple concrete objects, traditional flow-sensitive algorithms cannot kill the points-to information of field `f` of `o1` after the assignment statement at `L1` – it may unsoundly kill the points-to information of `r.f`. Hence the analysis concludes that `t1` may point to either `o2` or `o3` at the end of the program fragment (Figure 1(c)), although in any execution, `t1` can actually point to only `o3`. In general, `p` could have pointed to multiple abstract memory locations, which also would have made strong updates impossible for traditional flow-sensitive analyses.

In this paper we propose a different approach for flow-sensitive pointer analysis for Java programs that enables us to perform strong updates at indirect assignments effectively. Instead of a points-to graph, we compute a map from *access paths* to sets of abstract objects at each program statement. An access path is a local variable or a static field followed by zero or more field accesses. In the program fragment of Figure 1(a), the points-to set of the access path `p.f` can be strongly updated at `L1` regardless of whether `p` points to a single concrete memory location or not. On the other hand, the points-to set of `r.f` must be weakly updated at `L1` as `r.f` may alias to `p.f` at that program statement

[1] All analyses considered in this paper are *field-sensitive* [27].

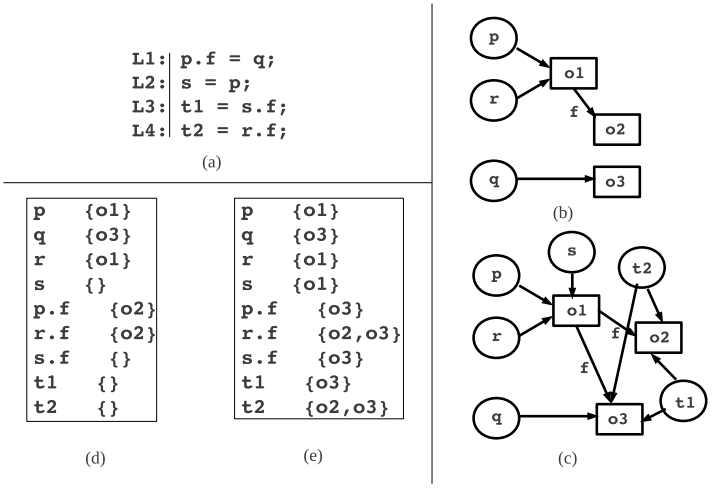

Fig. 1. (a) An example program fragment. (b) Points-to graph before the program fragment. (c) Points-to graph after the program fragment. (d) Our points-to information before the program fragment. (e) Our points-to information after the program fragment.

(two access paths may alias if they may refer to the same memory location). Note that we would strongly update `p.f` at `L1` even if `p` pointed to multiple abstract objects. We have observed that it is quite common to have (possibly interprocedural) program paths like Figure 1(a) which begin with an assignment to an access path and then subsequently read the access path, either directly or through an alias established by intervening pointer assignments. Our analysis targets such patterns and propagates the points-to sets from the initial assignment to the final read effectively through a series of strong updates.

While there has been earlier work on pointer analysis based on access paths for C programs [20,6], our approach differs from them in several ways. The key challenge to the scalability of such an analysis is the proliferation of access paths. In the presence of recursive data-structures, the number of access paths in a program can be infinite. As is standard, we bound the length of access paths using a user defined parameter l. The number of access paths however still grows exponentially with l, and it can be very expensive to maintain the full map of all access paths to their points-to sets at every program statement. One key feature of our algorithm is that we only need to store points-to sets of access paths that are *in scope* at a program statement. We also do further optimizations to reduce the size of the maps stored at each program statement as detailed in Section 3.3.

We bootstrap our analysis using a fast flow and context insensitive points-to analysis [1]. This base analysis is used in various stages of our analysis: to compute the set of access paths, to supply points-to sets of access paths longer than

the user-defined bound and to reach the fixpoint quickly by approximately precomputing the set of access paths modified at each program statement through aliasing.

We have implemented our analysis in the Chord framework [24]. The core of the analysis is written declaratively in Datalog [32]. Chord implements all Datalog relations using binary decision diagrams (BDD) [4] which helps in reducing the space required to store the points-to information. We have implemented our analysis both with and without context-sensitivity. Our analysis was run on eight moderately large Java programs with different values of l (the bound on the access path lengths) and we compared the precision of points-to sets and call-graphs with the traditional points-to graph based flow-insensitive [1] and flow-sensitive [16] analyses. On these benchmarks, for $l = 3$, our flow-sensitive and context-sensitive analysis shows a significant average improvement of 22% in precision over the flow-insensitive analysis with the same level of context-sensitivity, while terminating within reasonable time, whereas traditional flow-sensitive analysis has only less than 2% precision improvement over the flow-insensitive analysis and is much slower.

The rest of this paper is organized as follows. We give an overview of our technique with a couple of examples in Section 2. Section 3 describes our technique formally. We discuss an implementation of our technique and present empirical results in Section 4. Related works are discussed in Section 5. We discuss future directions and conclude with Section 6.

2 Overview

In this section, we informally describe the core of our algorithm using the program fragments of Figure 1(a) and Figure 2(a).

We first explain the intraprocedural part of our analysis using Figure 1. The intraprocedural analysis is an iterative dataflow analysis over the control flow graph (CFG). Our dataflow facts are maps from access paths to sets of abstract objects. We first compute a flow-insensitive points-to set for each variable. Let us assume that the points-to graph computed by the flow-insensitive analysis is as shown in Figure 1(c). The object `o1` has only one field, `f` and the objects `o2` and `o3` do not have any field. We also assume that the length of access paths is bound by the constant two. Hence the set of access paths in the program is {`p`, `q`, `r`, `s`, `p.f`, `r.f`, `s.f`, `t1`, `t2`}. Let us assume that the points-to information computed by our algorithm before `L1` is as shown in Figure 1(b). Figure 1(d) shows this information in our representation. The assignment at `L1` strongly updates the points-to set of `p.f` to {`o3`}. According to the flow-insensitive analysis, `r.f` may alias with `p.f` – hence the points-to set of `r.f` is updated to {`o2`,`o3`}. Although we could use our points-to information to detect the alias between `p.f` and `r.f` at `L1`, using a precomputed flow-insensitive analysis helps in reaching the fixpoint quickly. Note that this approximation may result in more weak updates, but does not affect strong updates. On the other hand, this approximation is

necessary for the scalability of our technique – without this approximation, six out of eight benchmarks did not terminate within 30 mins. Also note that `s.f` is not updated as it is not live at `L1`. The assignment at `L2` strongly updates the points-to sets of `s` and `s.f` with the points-to sets of `p` and `p.f` respectively. As in Java, a local variable like `s` cannot alias with any other access path, this assignment does not weakly update any access path. The assignments at `L3` and `L4` assigns the points-to sets of `s.f` and `r.f` to `t1` and `t2` respectively. The final points-to information is shown in Figure 1(e). The points-to set of `t1` in our analysis is more precise than the one computed by a traditional flow-sensitive analysis (Figure 1(c)).

We demonstrate the interprocedural analysis using the program fragment of Figure 2(a) (the statement at `LD` is commented out). In interprocedural analysis, at each program statement, we only store the information about access paths that are in scope at that statement. An access path is in scope at a statement if the access path starts with a variable local to the function containing the statement or with a static field. For the program fragment of Figure 2(a), the access paths of the outer function and the initial map at `LB` is shown in Figure 2(b). We use a mod/ref analysis based on the flow-insensitive points-to graph to determine that the points-to sets of `p.f` and `r.f` may be modified by the call to the function `setF`. On call to the function `setF`, `this`, `a`, and `this.f` are assigned the points-to sets of `p`, `q`, and `p.f` respectively. The assignment at `L8` strongly updates the points-to set of `this.f`. The map at `L9` is shown in Figure 2(c). On return to the outer function, as `p` and `this` must point to the same concrete object in all executions, the access path `p.f` can be strongly updated with the points-to set of `this.f`. On the other hand, as `r.f` may also be modified by the call to `setF` (because it is an alias of `p.f`) but `r` is not an actual parameter to the call, it is conservatively assigned its flow-insensitive points-to set. The map at `LC` is shown in Figure 2(d). The call to `getF` does not modify points-to sets of any access paths belonging to the outer function, but assigns the return variable of `getF` to `t1`. The final map is shown in Figure 2(e). Here also, the points-to set of `t1` is more precise than traditional flow-sensitive analyses which would not be able to do the strong update at `L8`.

If we include the statement at `LD`, the context-insensitive analysis would merge the points-to sets coming from `p.f` and `r.f` into the points-to set of `this.f` in function `getF`. This would make the points-to sets of both `t1` and `t2` to be {`o2`,`o3`}. Adding context-sensitivity would avoid this problem as two calls of `getF` would be distinguished by a context-sensitive analysis. For example, using a length 1 call-string as context would create two maps at `L3`, one tagged with call-site `LC` and mapping `this.f` to {`o3`} and another tagged with call-site `LD` and mapping `this.f` to {`o2`,`o3`}. On return, only the first map is used to assign points-to set of `t1`, making it {`o3`}. As Java programs use method calls extensively, we use call-string based context-sensitive analysis [30] to tag dataflow facts with fixed-length call-strings to distinguish between the calling contexts.

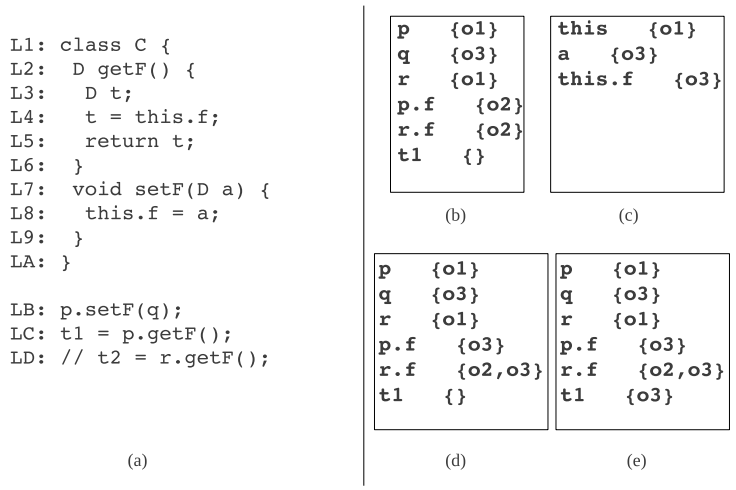

Fig. 2. (a) An example program fragment. (b) - (e) Points-to information at program statements `LB`, `L9`, `LC` and `LD`

3 Access Path-Based Flow-Sensitive Analysis

3.1 Background

Our input language is an intermediate code generated by the Chord framework [24] from Java bytecode. The intermediate language has at most one dereference per statement and is converted into partial *single static assignment* (SSA) form [11].

In SSA form, each variable is defined only once. If a variable is defined multiple times in the original program, each of those definitions is converted to a new version of that variable in SSA form. If multiple definitions of the same original variable reach a join point in the control flow graph, a ϕ statement is introduced at the join point. The ϕ statement merges the corresponding versions and creates a new version of the variable. In partial SSA form of Java programs, only the local variables are converted into SSA form. In a Java program, a local variable cannot be pointed by any variable or object field – hence there are no indirect assignments to these variables. Therefore, the definitions of such variables can be identified syntactically and can be converted to SSA form easily, without any prior pointer analysis.

Following are the different types of statements relevant to our pointer analysis and their representative forms. The variables `p`, `q`, `r` and the field `f` are of reference (pointer) type. `C` is the name of a class and `foo` is the name of a function. Without loss of generality, we consider function calls with only one parameter and ϕ statements merging two versions of a variable.

Allocation: `p = new C;`
Null assignment: `p = null;`
Copy: `p = q;`
Phi: `p =` ϕ`(q,r);`
Getfield: `p = q.f;`
Putfield: `p.f = q;`
Function call: `p = foo(q);`
Return: `return p;`

We do not consider static fields in this section. As array elements cannot be strongly updated in general, we treat array elements flow-insensitively. We assume that there is a `main` function designated as the entry point of the program. We also assume that all functions except the `main` function are virtual functions – the exact target function of a function call is determined by the type of the object pointed to by the first actual argument of the call. We also do not consider exceptions in this section. Our technique can easily be extended to handle all features of Java language and our implementation can take any Java program as input.

Like most other pointer analyses, we use allocation sites as abstract objects to represent all concrete objects allocated at that site. We represent the set of abstract objects in a program by *Objects*. In the rest of the paper, the word "object" means "abstract object", unless otherwise specified.

We bootstrap out analysis with a fast flow- and context-insensitive analysis [1]. This analysis computes the pointer information as a points-to graph G. We assume that the points-to graph supports two types of queries: $VarPts(G, v)$ returns the points-to set of the variable v and $ObjPts(G, o, f)$ returns the set of objects pointed to by the field f of the object o. Procedure 2 (*FIPts*) computes the set of objects pointed to by an access path in the points-to graph. The function *FIAlias*, defined below, detects if two different access paths may alias according to a points to graph G.

$$\begin{aligned} FIAlias_G = \lambda ap_1.\lambda ap_2. \quad & \text{if} \quad ap_1 \neq ap_2 \\ & \text{and} \quad ap_1 = ap_1'.f \text{ and } ap_2 = ap_2'.f \\ & \text{and} \quad FIPts(G, ap_1') \cap FIPts(G, ap_2') \neq \emptyset \\ & \qquad \text{then true else false.} \end{aligned}$$

3.2 Computing the Set of Access Paths

An *access path* is a local variable followed by zero or more field accesses[2]. More formally, given a program P with set of variables V and set of fields F, an access path is a member of the set $V(.F)^*$. The variable is called the *root* of the access path. The length of an access path is one more than the number of

[2] In general, an access path may start with a static field, we we do not consider static fields in this section for sake of simplicity.

Algorithm 1. Algorithm for computing set of access paths in a program

Input: Set of variables V, Points-to graph G, Map from abstract objects to actual types T, Bound on the lengths of access paths l.
Output: AP: Set of access paths in the function.

```
i = 1
AP ← V
newaps ← V
while i < k do
  nextaps ← ∅
  for all ap ∈ newaps do
    objset ← FIPts(G, ap)
  end for
  for all obj ∈ objset do
    objtype ← T(obj)
    for all field f of objtype do
      add ap.f to AP
      add ap.f to nextaps
    end for
  end for
  i ← i + 1
  newaps ← nextaps
end while
```

field accesses. In the presence of recursive data-structures, the set of access paths in a program can be infinite. We only consider access paths whose length is bound by a user-defined parameter l.

In order to compute the set of access paths in a program, we need to know that given an access path, which field accesses can be appended to it to generate longer access paths. It might be tempting to use the declared types of variables and fields to determine which field accesses are possible, but in the presence of inheritance, this approach runs into the following problem. Suppose `p` is a variable with declared type `A` and `q` is a variable with declared type `B`. Suppose `B` is a subtype of `A` and the class `B` has a field `f` which is not present in `A`. Suppose further that there is an assignment `p = q`. As there is no access `p.f`, we lose the points-to information of field `f` of the object pointed to by `p` after the assignment. Again, if there is downcast `q = (B)p`, we cannot determine the points-to information of `q.f` after the assignment. To avoid this problem, we use Algorithm 1 to compute the set of access paths in a program. This algorithm uses a flow and context insensitive points-to analysis to compute a points-to graph for the entire program. Given an access path, we traverse the points-to graph to determine the objects pointed to by the access path (Procedure 2). We extend the access path with all the fields of the actual types of these objects.

3.3 Intraprocedural Analysis

In this section, we assume that the input program has a single function with no call statements in order to focus on the intraprocedural analysis. Given a

Procedure 2. *FIPts*

Input: Points-to graph G, Access path ap.
Output: *objset*: Set of objects pointed to by ap in G.

```
if ap is a variable then
    objset ← VarPts(G, ap)
else if ap is of the form ap'.f then
    objset ← ⋃_{o ∈ FIPts(ap')} ObjPts(G, o, f)
end if
return objset
```

program function P, our intraprocedural analysis is an instance of iterative dataflow analysis [19] over the CFG $C = (N, E)$ of P, where N is the set of nodes of the CFG, representing the statements of the function and E is the set of control flow edges. We denote the root node of the CFG by n_0 and predecessor of a node n by $pred(n)$. The dataflow analysis $\mathcal{D} = (\mathcal{L}, \mathcal{F})$ of the function consists of a lattice $\mathcal{L}$ and a set of transfer functions $\mathcal{F}$, defined below.

Dataflow Lattice: The dataflow lattice $\mathcal{L} = (\mathcal{M}, \preceq)$ consists of a set of dataflow facts $\mathcal{M}$ and an ordering relation $\preceq$. The set $\mathcal{M}$ is the set of all maps from access paths to sets of abstract objects. Given two maps $m_1, m_2 \in \mathcal{M}$, $m_1 \preceq m_2$ iff for all access paths ap in AP, $m_1(ap) \subseteq m_2(ap)$. Naturally, the induced join operation is the point-wise union of points-to sets of all access paths in AP. Formally,

$$m_1 \sqcup m_2 = \lambda ap.\ (m_1(ap) \cup m_2(ap)).$$

Similarly, the greatest element of $\mathcal{M}$ is defined as $\top = \lambda ap.Objects$ and the least element as $\bot = \lambda ap.\emptyset$. As the sets AP and *Objects* are both finite, the set $\mathcal{M}$ is also finite.

Transfer Functions: The transfer function for a CFG node describes how the statement at that node modifies a dataflow fact. Given a node n and an input map m_{in}, the output map m_{out} might map some access paths to different points-to sets than m_{in}. Table 1 describes, for each type of statement mentioned in Section 3.1 except for call and return statements, which access paths are mapped differently in m_{out} compared to m_{in}. For all other access paths ap, $m_{out}(ap) = m_{in}(ap)$. For all statements not listed in Table 1, $m_{out} = m_{in}$. In the table, a is an arbitrary non-empty field access sequence of the form $F(.F)^*$ where F is the set of fields in the program.

If a variable `p` is assigned a new object `o`, the points-to set of `p` contains only `o` and other access paths with `p` as root do not point to any object after the assignment. If the lhs of an assignment is a variable (say `p`) and the rhs is an access path (say ap), points-to sets of `p` and all access paths of the form `p.`a (a is any non-empty field sequence) are strongly updated with the points-to sets of ap and $ap.a$, respectively. For Getfield statements, as the length of access path

Table 1. Intraprocedural transfer functions. Column 1 lists types of statements. Column 2 lists the access path ap for which $m_{out}(ap)$ is different from $m_{in}(ap)$. Column 3 defines the points-to sets of m_{out} for such access paths. Here a is a non-empty field access sequence of the form $F(.F)^*$.

Statement	ap	$m_{out}(ap)$
`//abstract object o` `p = new C`	`p`	{`o`}
	`p.`a	$\emptyset$
`p = null`	`p`	$\emptyset$
	`p.`a	$\emptyset$
`p = q`	`p`	m_{in}(`q`)
	`p.`a	m_{in}(`q.`a)
`p =` ϕ`(q, r)`	`p`	m_{in}(`q`) $\cup$ m_{in}(`r`)
	`p.`a	m_{in}(`q.`a) $\cup$ m_{in}(`r.`a)
`p = q.f`	`p`	m_{in}(`q.f`)
	`p.`a	if `q.f.`$a \in AP$ then m_{in}(`q.f.`a) else $FIPts(G,$ `q.f.`$a)$
`p.f = q`	`p.f`	m_{in}(`q`)
	`p.f.`a	m_{in}(`q.`a)
	ap'`.f`	if $FIPts(G, ap') \cap FIPts(G,$ `p`$) \neq \emptyset$ then m_{in}(`q`) $\cup$ $m_{in}(ap'$`.f`$)$ else $m_{in}(ap'$`.f`$)$
	ap'`.f.`a	if $FIPts(G, ap') \cap FIPts(G,$ `p`$) \neq \emptyset$ then m_{in}(`q.`a) $\cup$ $m_{in}(ap'$`.f.`$a)$ else $m_{in}(ap'$`.f.`$a)$

on the lhs (say `p`) is shorter than the length of the access path on the rhs (say `q.f`), there might exist some access path of the form `p.`a such that there is no access path `q.f.`a, as its length might be more than the user-specified bound. In such cases, we use the flow-insensitive analysis to supply the points-to set for `p.`a. For all these statements, only access paths with `p` as root need to be updated. On the other hand, for Putfield statements, the lhs and its extensions are strongly updated, whereas the aliases of the lhs and their extensions are weakly updated. Note that, instead of our own analysis, we use a precomputed flow and context insensitive pointer analysis to detect these aliases. Using our analysis to detect these aliases would cause our analysis to find new access path assignments during the fixpoint computation. On the other hand, using a base pointer analysis enables us to precompute the set of direct and indirect access path assignments at all program statements. This approximation speeds up the fixpoint computation significantly. Note that this approximation may cause our analysis to perform more weak updates, but strong updates are not affected.

Multiple field accesses: Although the intermediate language described in Section 3.1 has at most one field access per statement, the original Java program

may have multiple field accesses per statement. For example, the statement `p.f.g = q;` is converted to the following sequence of statements: `t1 = p.f; t1.g = q;`, where `t1` is a temporary variable, defined only once. After the assignment to `t1.g`, according to the rules in Table 1, the access path `p.f.g` should be weakly updated, although according to the original program, it could have been strongly updated. As `t1` is defined only once in the partial SSA form and it is used immediately after the definition, we know that `t1` must point to the same object as `p.f` before the second assignment in the intermediate code. Hence we strongly update `t1.g` as well as `p.f.g` at the second statement.

Dataflow Equations: The dataflow fact at n_0 is $\perp$ and the transfer function for node n is denoted by F_n. We compute a dataflow fact at each node of the CFG. More specifically, we compute the least solution for X for the following set of dataflow equations:

$$\begin{aligned} X[n_0] &= \perp \\ \forall n \in (N - \{n_0\}) : X[n] &= \bigsqcup_{n' \in pred(n)} F_{n'}(X[n']) \end{aligned} \tag{1}$$

Optimizations: Although the set AP is finite, it can be very large – growing exponentially with the length of the access paths. This large size of AP in turn increases the space consumed by the elements of the set $\mathcal{M}$ as well as increases the time to compute the transfer functions. In order to reduce the sizes of dataflow facts, we perform two optimizations, described below.

In partial SSA form, each local variable is defined only once. Hence, in our analysis, the points-to set of a local variable can be changed by only one statement – the one defining it. Hence, instead of maintaining the map from local variables to its points-to set at every program point, we maintain a single map from local variables to their points-to sets in each function. The dataflow facts at each program point are maps from access paths with length greater than one to their points-to sets. This technique is adopted from [11].

We also observe that the points-to information of an access path is typically useful only at the program statements where it is *live*. A variable is live at a program statement if its definition reaches that statement and the variable is used subsequently in the function. An access path is live at a program statement if the root variable is live at that statement. At a program statement, we only maintain the map from live access paths to their points-to sets. As there is a global map for variables, the points-to information of variables are sound at all program statements, irrespective of whether the variable is live at that statement or not.

3.4 Interprocedural Analysis

The interprocedural analysis is an iterative dataflow analysis over the interprocedural control flow graph (ICFG), constructed by taking disjoint union of

individual control flow graphs of all functions and then adding call and return edges. A *call edge* connects a call statement to the first statement of the called function and a *return edge* connects a return statement to the statements following the corresponding call statements. Call and return edges for virtual functions are added *on-the-fly*; at a call site, if the receiver variable (the first actual parameter in our intermediate language) points to an abstract object `o`, the actual function to be called is determined by the actual type of `o`. As new objects are added to the points-to set of the receiver variable, new call and return edges are added – fixpoint is reached when no new objects are added to any points-to set *and* no new call/return edges are added to the ICFG. For the sake of simplicity, in this section we assume that the call and return edges are added a priori.

Context-Insensitive Analysis: We first consider context-insensitive analysis where dataflow facts are not distinguished by the calling contexts. The dataflow facts are the same as the intraprocedural analysis – maps from access paths to sets of abstract objects – but instead of maintaining the map for all access paths at all program statements, we only maintain the map for access paths that are in scope at that statement. An access path ap is *in scope* at a statement s if the root of the access path is a variable local to the function containing s[3]. The statement s may modify the points-to set of an access path ap' not in scope at s. As our analysis does not store the points-to information of ap' at s, the change in the points-to set of ap' is not reflected immediately after s – it is updated on return to the function where ap' is in scope. The ability to discard the points-to information of access paths at program statements where they are not in scope but still soundly update them on return is the key to the scalability of our analysis.

Call statements in the ICFG have multiple outgoing edges – one edge to the next statement of the same function and (potentially multiple) call edges. Similarly, the return statements may also have multiple outgoing edges – one return edge for each calling method. The transfer functions for call and return statements propagate different dataflow information along different edges. Note that the first statement of a function acts as a join node in the ICFG, joining the call edges from different call-sites. In an ICFG, the statement following a call statement in the same function also joins the return edges from the called functions with the CFG edge. The transfer functions for all statements described in Section 3.3 remain unchanged, but to maintain uniformity with the call and return statements, we add an outgoing edge as a second parameter to the transfer function. For all statements n except for call and return, $F_n(m, e)$ is same as $F_n(m)$ as defined in Section 3.3, where e is an outgoing edge of node n and m is a dataflow fact.

Table 2 defines the transfer function for a call statement `foo(q)` along the CFG edge and along a call edge to the function `foo` with formal parameter `p` (as before, a denotes an arbitrary non-empty field sequence). Along the call edge,

[3] An access path with a static field as root is in scope everywhere, but we do not consider static fields in this section.

it initializes the points-to sets of access paths that have formal parameters of the called function as roots. Along the CFG edge, it kills the points-to sets of access paths that may be updated inside the called function. We use a mod/ref analysis based on the flow-insensitive points-to graph to determine if an access path may be modified by a called function. A function `foo` *may modify* a field `f` of an abstract object `o` if one of the following is true:

1. There is a statement in `foo` writing to `v.f` such that $\texttt{o} \in \mathit{FIPts}(G, v)$ (G is the flow-insensitive points-to graph).
2. `foo` calls a function which may modify the field `f` of the object `o`.

An access path l`.f` may be modified by a call to `foo` if $o \in \mathit{FIPts}(G, l)$ and `foo` may modify field `f` of object `o`. If an access path ap may be modified by a call to `foo`, we write $\mathit{MayMod}(\texttt{foo}, ap)$. Note that the output map along CFG edge only contains access paths local to the calling function, whereas the output map along the call edge only has access paths local to the called function.

Table 2. Transfer function for call statement `foo(q)` along the call edge to function `foo` with formal parameter `p` and along the CFG edge. Here a is an arbitrary non-empty field sequence.

Edge	ap	$m_{out}(ap)$
Call edge	`p`	$m_{in}(\texttt{q})$
	`p.`a	$m_{in}(\texttt{q}.a)$
	other	$\emptyset$
CFG edge	ap s.t. $\mathit{MayMod}(\texttt{foo}, ap)$	$\emptyset$
	other	$m_{in}(ap)$

Table 3 describes the transfer function for the return statement `return r` along the return edge corresponding to the call statement `s = foo(q)`. The formal parameter of the function `foo` containing the return statement is `p`. The output map only contains access paths local to the calling function. The return statement updates the points-to sets of the access paths rooted at `s` with the points-to sets of the corresponding access paths rooted at `r`.

We use the transfer function of the return statement to soundly update the points-to sets of access paths of the calling function that could have been modified during the execution of `foo`. As the input language is in partial SSA form, the formal parameter of a function cannot be reassigned inside the function. Hence, in every execution, the formal parameter `p` and the actual parameter `q` must point to the same concrete object throughout the execution of `foo`. Therefore, on return, the value of `q.`a would be same as the value of `p.`a at the return statement. Hence, if `q.`a may be modified by `foo`, the transfer function of the return statement assigns the points-to sets of access paths rooted at `p` to the points-to sets of the corresponding access paths rooted at `q`. As the points-to sets of access paths that may be modified by the called function are killed at the

call statement (Table 2), this results in strong updates of such access paths after the join of the return edge with the CFG edge of the calling function. Any access paths that can be modified by `foo` but not rooted at the actual parameter of the call are assigned their flow-insensitive points-to set after the call statement. Any access paths that cannot be modified by `foo` are assigned empty sets. As the points-to sets of such access paths are not killed by the call statements, they retain their points-to set prior to the call to `foo` after the join with the CFG edge.

Table 3. Transfer function for return statement `return r` along return edge corresponding to the call statement `s = foo(q)`. The formal parameter of the function `foo` containing the return statement is `p`. Here G is a points-to graph computed by the base analysis and a is an arbitrary non-empty field sequence.

ap	$m_{out}(ap)$
`s`	$m_{in}(\texttt{r})$
`s.`a	$m_{in}(\texttt{r}.a)$
`q.`a	if $MayMod(\texttt{foo}, ap)$ then $m_{in}(\texttt{p}.a)$ else $\emptyset$
other	if $MayMod(\texttt{foo}, ap)$ then $FIPts(G, ap)$ else $\emptyset$

The transfer function for return statement can be imprecise for many access paths; but it can perform strong updates for access paths rooted at the actual parameters. In Java programs, often such access paths are read later, either directly or through an alias established though intervening pointer assignments. Our technique can have the benefit of strong updates in such cases.

The interprocedural dataflow analysis computes the least solution of the following set of dataflow equations. Here n_0 is the root node of the ICFG, i.e. the root node of the `main` function and $\langle n', n\rangle$ denotes the ICFG edge between nodes n' and n.

$$\begin{aligned} X[n_0] &= \bot \\ \forall n \in (N - \{n_0\}) : X[n] &= \bigsqcup_{n' \in pred(n)} F_{n'}(X[n'], \langle n', n\rangle) \end{aligned} \tag{2}$$

Context-Sensitive Analysis: For context-sensitivity, we use the standard call-string approach [30] with finite sequence of call-sites as contexts.

We first describe the context-sensitive technique with unbounded call-strings. Let $\mathcal{D} = (\mathcal{L}, \mathcal{F})$ be the underlying dataflow analysis with $\mathcal{L} = (\mathcal{M}, \preceq)$. Let $C^* = (N, E)$ denote the ICFG of the program. We define a *call-string* γ as a

(possibly empty) sequence of call statements. Let Γ be the set of all such call-strings. The empty call-string is denoted by ϵ. The length of a call-string γ is denoted by $|\gamma|$. The ith component of γ is denoted by $\gamma[i]$ and the substring from ith to jth component (both inclusive) is denoted by $\gamma[i..j]$. The operator "$\cdot$" denotes the string append operation.

The call-string approach defines a new dataflow analysis framework $\mathcal{D}^* = (\mathcal{L}^*, \mathcal{F}^*)$, where $\mathcal{L}^* = (\mathcal{M}^*, \preceq^*)$. The domain $\mathcal{M}^*$ is the space of all maps from Γ into $\mathcal{M}$. The ordering in $\mathcal{L}^*$ is the point-wise ordering on $\mathcal{L}$, i.e. for $\xi_1, \xi_2 \in \mathcal{M}^*, \xi_1 \preceq^* \xi_2$ iff $\forall \gamma \in \Gamma, \xi_1(\gamma) \preceq \xi_2(\gamma)$.

In order to define the flow functions, we first define a partial binary operator $\circ : \Gamma \times E \to \Gamma$ in the following way:

$$\gamma \circ \langle n, n' \rangle = \begin{cases} \gamma \cdot n & \text{if } \langle n, n' \rangle \text{ is a call edge} \\ \gamma[1..|\gamma| - 1] & \text{if } \langle n, n' \rangle \text{ is a return edge and } \gamma[|\gamma|] \text{ is the corresponding call statement} \\ \gamma & \text{otherwise} \end{cases}$$

A flow function $F_n^* \in \mathcal{F}^*$, where $n \in N$, is a function from $\mathcal{M}^* \times E$ to $\mathcal{M}^*$, defined below:

$$F_n^*(\xi, e)(\gamma) = \begin{cases} F_n(\xi(\gamma'), n) & \text{if there exists a unique } \gamma' \text{ such that } \gamma = \gamma' \circ e \\ \bot & \text{otherwise} \end{cases}$$

The solution of the analysis is the least solution of the dataflow equations corresponding to the lattice and transfer functions defined above.

As the set of unbounded call-strings is infinite, we use a k length suffix of the unbounded call-string as approximate call-string. Details of the call-string approach can be found in [30].

4 Implementation and Experimental Results

We have implemented our analysis within the Chord framework [24]. Chord encodes program structures such as CFG, assignment statements and type hierarchies as relations and implements them using BDDs [4]. We use Chord's built-in flow and context insensitive pointer analysis as our base analysis. Our frontend, written in Java, computes the set of access paths and other relevant program information as relations implemented using BDDs. The core analysis is written declaratively in Datalog [32] which takes the relations produced by the frontend as input. Our implementation first converts each assignment of the program into multiple assignments to access paths, capturing all possible strong and weak updates of access paths by that assignment. We use the precomputed base pointer analysis to perform the possible weak updates. The next phase of the analysis computes the flow-sensitive points-to sets for access paths and constructs the call-graph on-the-fly. Chord uses `bddbddb` [36] for fixpoint computation of the Datalog analyses. We have implemented our analysis both with and without context-sensitivity. The context-sensitive analysis is a call-string analysis with call-string length 1.

Table 4. Characteristics of the benchmarks

Benchmarks	Int. Code Stmts	Classes	Methods	AP($l = 2$)	AP($l = 3$)	Contexts
polyglot	123789	1474	5933	14651	113354	24690
jlex	126661	1411	5708	21149	137578	10037
javacup	117804	1389	5498	22018	136773	11721
jtopas	120499	1432	5736	17308	137714	9068
jdom	119437	1443	5610	13704	107536	8473
jasmin	123490	1381	5174	39480	179780	11307
jjdoc	85256	1270	3701	31001	63540	9485
jjtree	93958	1376	4219	32343	58018	11155

We have run our analysis on eight moderately large Java benchmarks. We have used a laptop with 2.3 GHz Core i5 processor with 3GB memory for our experiments. We have used OpenJDK 1.6 as our JDK. Table 4 shows the sizes of intermediate code, number of classes and methods, number of access paths with bound 2 and 3 and the number of contexts for these benchmarks. As our analysis includes the Java libraries as well, we report the number of lines in the intermediate code within the scope of the analysis as constructed by Rapid Type Analysis, instead of the size of source code of the application program only. Note that the number of access paths grows rapidly with respect to l.

We compare the precision of our analysis with that of the points-to graph based flow-insensitive [1] and flow-sensitive [16] analyses. As measurement of precision, we use the sizes of the points-to sets of the local variables. Note that due to the partial SSA form, our flow-sensitive analysis stores a single points-to set for each local variable for the entire program (cf. Section 3.3). Hence, we can directly compare the total size of the points-to sets of local variables obtained by our analysis with that of the points-to graph based flow-insensitive and flow-sensitive analyses. Note that the heap-based memory locations are represented differently in our analysis than points-to graph based analyses; hence we do not compare the sizes of such memory locations directly. Nevertheless, as our intermediate code is single-dereference based, contents of any heap location must be copied to a local variable before it can be dereferenced. Thus any change in the sizes of the points-to sets of heap locations are reflected in the sizes of the points-to sets of the local variables.

We also construct call-graphs of the input programs using the pointer information. Nodes of a call-graph are methods of the input program. If a method `m` calls a method `n`, the call-graph has an edge from `m` to `n`. For virtual calls, the actual method to be called at run-time depends on the object pointed to by the first actual parameter of the call site – hence a precise pointer analysis may reduce the number of edges of the call-graph (henceforth we refer to the number of edges of a call-graph as its *size*).

We observe that the precision improvement of our analysis over traditional flow-insensitive analysis without context-sensitivity is very small – only 5% on average for points-to sets of local variables and 6% on average for call-graphs.

This is expected for Java programs as they use method calls extensively to access fields and without context-sensitivity, the points-to sets of access paths in those methods are merged for all calls. Such a situation is shown in Section 2.

With a call-string length of 1, our flow- and context-sensitive analysis shows significant precision improvement over the flow-insensitive analysis with the same level of context-sensitivity. The flow-insensitive analysis also uses the partial SSA form, hence it already has the benefit of flow-sensitivity for local variables [11] – our analysis shows precision improvements on top of that. On the other hand, the points-to graph based flow-sensitive analysis shows only less than 2% improvement over the flow-insensitive analysis. Hence, the precision improvement of our analysis comes only from strong updates of heap-residing pointers. Figure 3 shows the precision improvements for points-to sets of local variables over the flow-insensitive analysis on eight benchmarks for different bounds on the lengths of the access paths. It also shows the result for the traditional points-to graph based flow-sensitive analysis. On the average, our flow-sensitive analysis reduces the sizes of points-to sets of local variables by 22% with $l = 3$ and by 16% with $l = 2$ over the flow-insensitive analysis with partial SSA form. All analyses are context-sensitive with call-string length 1.

We also report the reduction in call-graph size over the flow-insensitive analysis in Figure 4. The average reduction in call-graph size is 30% for $l = 3$ and 26% for $l = 2$. Table 5 shows the time taken by the flow-insensitive analysis, the points-to graph based flow-sensitive analysis and our flow-sensitive analysis (for $l = 2$ and $l = 3$) on these benchmarks. On the other hand, the traditional flow-sensitive analysis does not show any non-trivial reduction in the call-graph sizes with respect to the flow-insensitive analysis – hence we omit the comparison with such analysis in Figure 4 for the sake of clarity.

On the average, our analysis has a slowdown of 9.2X with $l = 3$ and of 5.8X with $l = 2$ with respect to the flow-insensitive analysis, but it is much faster than the points-to graph based flow-sensitive analysis.

Table 5. Time taken by flow-insensitive analysis (FI time), points-to graph based flow-sensitive analysis and our our flow-sensitive analysis (with access path length 2 and 3). All analyses are context-sensitive with call-string length 1.

Benchmarks	FI time	FS time (points-to graph)	FS time ($l = 2$)	FS time ($l = 3$)
polyglot	52	756	201	411
jlex	51	858	220	421
javacup	61	838	477	808
jtopas	47	820	306	390
jdom	49	609	297	378
jasmin	58	824	388	552
jjdoc	36	589	166	286
jjtree	44	610	228	424
Average	**49.8**	**738.0**	**292.9**	**458.8**

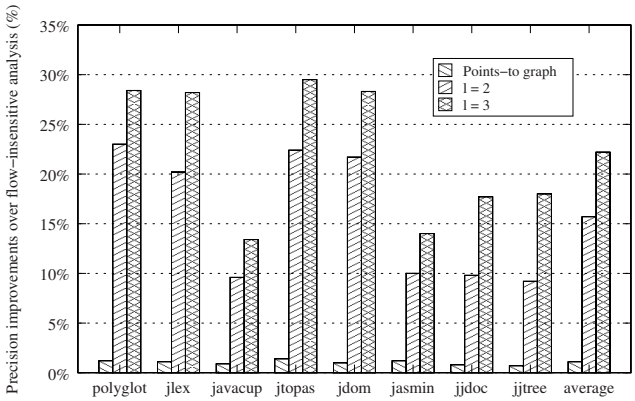

Fig. 3. Reduction in sizes of points-to sets by points-to graph based flow-sensitive analysis and our flow-sensitive analysis (with access path lengths 2 and 3) over flow-insensitive analysis with partial SSA form. All analyses are context-sensitive with call-string length 1.

These empirical results show that our analysis has significant precision improvement over the flow-insensitive analysis due to the strong updates of heap-based pointers which can not be achieved by traditional flow-sensitive analysis. The time taken by our analysis is also reasonable compared to the traditional flow-sensitive analysis.

5 Related Work

Flow-Sensitive Pointer Analysis: Pointer analysis is a fundamental static analysis with a long history. Early flow-sensitive pointer analyses [20,6] explicitly stored the pairs of access paths that might alias with each other. These works do not focus on dynamically allocated data-structures. The experimental results are preliminary. Emami et al. [7] presents a flow-sensitive and context-sensitive pointer analysis that uses points-to information between abstract stack locations as dataflow facts. They mark each points-to relation as *may* or *must* to perform strong updates on indirect assignments. Some analyses [37,35] use an intraprocedural flow-sensitive analysis to build procedure summaries that are instantiated at call-sites, but none of these analyses can perform strong updates on pointers residing in the heap. Hasti and Horwitz [13] incrementally build an SSA representation from the aliases already discovered – a flow-insensitive analysis on the SSA form gives the same benefit as a flow-sensitive one for the memory location already converted into SSA form. It remains an open question whether the fixpoint of this technique matches the result of a flow-sensitive analysis. Hind et al. [15] express the flow-sensitive pointer analysis as an iterative

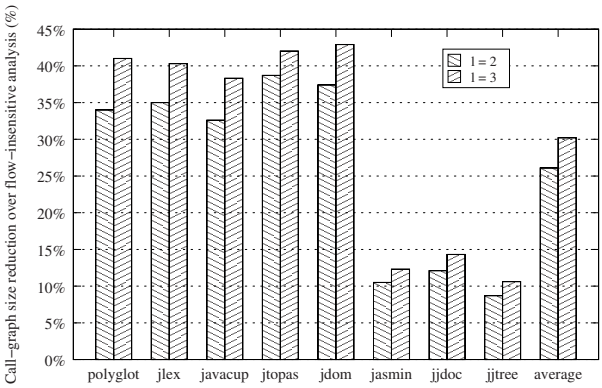

Fig. 4. Reduction in call-graph sizes by our flow-sensitive analysis (with access path lengths 2 and 3) over flow-insensitive analysis with partial SSA form. All analyses are context-sensitive with call-string length 1.

dataflow analysis over the CFG. They use a *compact representation*, essentially a form of points-to graph, as dataflow facts. In order to perform strong updates at indirect assignments, they keep track of whether a pointer points to a single concrete object.

More recently, researchers have focused on improving the scalability of flow-sensitive pointer analysis for C programs. Hardekopf and Lin [11] proposed a semi-sparse analysis which uses a partial SSA form for top-level variables and a sparse evaluation graph to eliminate irrelevant nodes. This approach is further extended in [12], where a flow-insensitive analysis is used to compute approximate def-use pairs, which helps in speeding up the sparse analysis in a later stage. The technique proposed by Yu et al. [38] first partitions the pointers into different levels by a flow-insensitive analysis such that there is an unidirectional flow of value from higher to lower level. Once the higher level variables are analyzed, the result can be used to build SSA representation for the lower level variables. Lhotak et al. [21] performs flow-sensitive analysis only on those memory locations which can be strongly updated. Li et al. [22] reduce the flow-sensitive pointer analysis problem to a graph reachability problem in a value flow graph which represents dependence between pointer variables. All these analyses do not perform strong updates for heap-residing pointers.

Zhu [39] uses BDDs to improve scalability of flow and context sensitive pointer analysis. This technique cannot perform any strong updates as querying whether a variable points to a single object is not efficiently supported by BDDs. As our technique does not need such uniqueness queries to perform strong updates, we can use BDDs efficiently.

Fink et al. [8] proposes a flow and context sensitive analysis for typestate verification. They use access paths to determine if a concrete object is *live*, i.e.

accessible via some access paths. They use a uniqueness analysis to identify abstract objects that represents a single live concrete object so that strong updates can be applied to those objects. It is not known how many strong updates can be done for general pointer analysis using their technique. Our analysis does not rely on the uniqueness of an abstract object to perform strong updates.

Bootstrapping: Several pointer analysis techniques use a fast and imprecise analysis to bootstrap their own analysis. Kahlon [18] uses a fast and imprecise analysis to partition the code such that each part can be analyzed independently. Similarly, Fink et al. [8] apply successively more precise techniques to smaller parts of the code. As mentioned before, Yu et al. [38] uses a flow-insensitive analysis to partition the pointers into different levels. Similarly, Hardekopf et al. [12] uses an auxiliary flow and context insensitive analysis to compute the approximate def-use chains.

Declarative Pointer Analysis: Whaley [33] developed `bddbddb`, a framework for implementing program analyses declaratively in Datalog [32] and implemented a context-sensitive pointer analysis using this framework. Later, Bravenboer and Smaragdakis [3] implemented several context-sensitive analyses in Datalog. All these algorithms are flow-insensitive.

6 Conclusion and Future Work

In this paper, we have presented a flow-sensitive pointer analysis algorithm for Java that can perform strong updates on pointers residing in the heap. Our implementation scales for moderately large benchmarks. Our flow and context sensitive analysis shows significant precision improvement over the flow-insensitive analysis with partial SSA form as well as traditional points-to graph based flow-sensitive analysis, with same level of context-sensitivity, on those benchmarks.

In future, we would like to improve the scalability of our analysis further by implementing it over sparse evaluation graphs [11]. We would also like to incorporate different types of context-sensitivity in our analysis such as object-sensitivity [23].

References

1. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, University of Copenhagen (1994)
2. Berndl, M., Lhoták, O., Qian, F., Hendren, L., Umanee, N.: Points-to analysis using bdds. In: PLDI 2003: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, pp. 103–114. ACM, New York (2003)
3. Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. In: Proceeding of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2009, pp. 243–262. ACM, New York (2009)

4. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Comput. 35, 677–691 (1986)
5. Chang, W., Streiff, B., Lin, C.: Efficient and extensible security enforcement using dynamic data flow analysis. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS 2008, pp. 39–50. ACM, New York (2008)
6. Choi, J.-D., Burke, M., Carini, P.: Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1993, pp. 232–245. ACM, New York (1993)
7. Emami, M., Ghiya, R., Hendren, L.J.: Context-sensitive interprocedural points-to analysis in the presence of function pointers. In: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, PLDI 1994, pp. 242–256. ACM, New York (1994)
8. Fink, S.J., Yahav, E., Dor, N., Ramalingam, G., Geay, E.: Effective typestate verification in the presence of aliasing. ACM Trans. Softw. Eng. Methodol. 17, 9:1–9:34 (2008)
9. Guyer, S.Z., Lin, C.: Error checking with client-driven pointer analysis. Sci. Comput. Program. 58, 83–114 (2005)
10. Hardekopf, B., Lin, C.: The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. SIGPLAN Not. 42(6), 290–299 (2007)
11. Hardekopf, B., Lin, C.: Semi-sparse flow-sensitive pointer analysis. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, pp. 226–238. ACM, New York (2009)
12. Hardekopf, B., Lin, C.: Flow-sensitive pointer analysis for millions of lines of code. In: 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 289–298 (April 2011)
13. Hasti, R., Horwitz, S.: Using static single assignment form to improve flow-insensitive pointer analysis. In: PLDI 1998: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, pp. 97–105. ACM, New York (1998)
14. Heintze, N., Tardieu, O.: Ultra-fast aliasing analysis using cla: a million lines of c code in a second. In: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, PLDI 2001, pp. 254–263. ACM, New York (2001)
15. Hind, M., Burke, M., Carini, P., Choi, J.-D.: Interprocedural pointer alias analysis. ACM Trans. Program. Lang. Syst. 21, 848–894 (1999)
16. Hind, M., Pioli, A.: Assessing the Effects of Flow-Sensitivity on Pointer Alias Analyses. In: Levi, G. (ed.) SAS 1998. LNCS, vol. 1503, pp. 57–81. Springer, Heidelberg (1998)
17. Hind, M., Pioli, A.: Which pointer analysis should i use? In: ISSTA 2000: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 113–123. ACM, New York (2000)
18. Kahlon, V.: Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In: PLDI 2008: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 249–259. ACM, New York (2008)

19. Kildall, G.A.: A unified approach to global program optimization. In: POPL 1973: Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 194–206. ACM, New York (1973)
20. Landi, W., Ryder, B.G.: A safe approximate algorithm for interprocedural aliasing. In: PLDI 1992: Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation, pp. 235–248. ACM, New York (1992)
21. Lhoták, O., Chung, K.-C.A.: Points-to analysis with efficient strong updates. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, pp. 3–16. ACM, New York (2011)
22. Li, L., Cifuentes, C., Keynes, N.: Boosting the performance of flow-sensitive points-to analysis using value flow. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, SIGSOFT/FSE 2011, pp. 343–353. ACM, New York (2011)
23. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for java. ACM Trans. Softw. Eng. Methodol. 14, 1–41 (2005)
24. Naik, M.: jchord: A static and dynamic program analysis platform for java, `http://code.google.com/p/jchord/`
25. Nystrom, E.M., Kim, H.-S., Hwu, W.-m.W.: Bottom-Up and Top-Down Context-Sensitive Summary-Based Pointer Analysis. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 165–180. Springer, Heidelberg (2004)
26. Pearce, D.J.: Some directed graph algorithms and their application to pointer analysis. PhD thesis, University of London, Imperial College of Science, Technology and Medicine, Department of Computing (2005)
27. Pearce, D.J., Kelly, P.H., Hankin, C.: Efficient field-sensitive pointer analysis of c. ACM Trans. Program. Lang. Syst. 30(1) (November 2007)
28. Salcianu, A., Rinard, M.: Pointer and escape analysis for multithreaded programs. In: PPoPP 2001: Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming, pp. 12–23. ACM, New York (2001)
29. Shapiro II, M., Horwitz, S.: The Effects of the Precision of Pointer Analysis. In: Van Hentenryck, P. (ed.) SAS 1997. LNCS, vol. 1302, pp. 16–34. Springer, Heidelberg (1997)
30. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis, ch.7, pp. 189–234. Prentice-Hall, Englewood Cliffs (1981)
31. Steensgaard, B.: Points-to analysis in almost linear time. In: POPL 1996: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 32–41. ACM, New York (1996)
32. Ullman, J.D.: Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies. W. H. Freeman & Co., New York (1990)
33. Whaley, J.: Context-Sensitive Pointer Analysis using Binary Decision Diagrams. PhD thesis, Stanford University (March 2007)
34. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, PLDI 2004, pp. 131–144. ACM, New York (2004)
35. Whaley, J., Rinard, M.: Compositional pointer and escape analysis for java programs. In: OOPSLA 1999: Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, pp. 187–206. ACM, New York (1999)
36. Whaley, J., Unkel, C., Lam, M.S.: A bdd-based deductive database for program analysis (2004), `http://suif.stanford.edu/bddbddb`

37. Wilson, R.P., Lam, M.S.: Efficient context-sensitive pointer analysis for c programs. In: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation, PLDI 1995, pp. 1–12. ACM, New York (1995)
38. Yu, H., Xue, J., Huo, W., Feng, X., Zhang, Z.: Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In: Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2010, pp. 218–229. ACM, New York (2010)
39. Zhu, J.: Towards scalable flow and context sensitive pointer analysis. In: Proceedings of the 42nd Annual Design Automation Conference, DAC 2005, pp. 831–836. ACM, New York (2005)