Demand-Driven Context-Sensitive Alias Analysis for Java

Dacong Yan Guoqing Xu Atanas Rountev
Ohio State University
{yan,xug,rountev}@cse.ohio-state.edu

ABSTRACT

Software tools for program understanding, transformation, verification, and testing often require an efficient yet highly-precise alias analysis. Typically this is done by computing points-to information, from which alias queries can be answered. This paper presents a novel context-sensitive, demand-driven alias analysis for Java that achieves efficiency by answering alias queries *directly*, instead of relying on an underlying points-to analysis. The analysis is formulated as a context-free-language (CFL) reachability problem over a language that models calling context sensitivity, and over another language that models field sensitivity (i.e., flow of reference values through fields of heap objects).

To improve analysis scalability, we propose to compute *procedural reachability summaries* online, during the CFL-reachability computation. This cannot be done indiscriminately, as the benefits of using the summary information do not necessarily outweigh the cost of computing it. Our approach selects for summarization only a subset of heavily-used methods (i.e., methods having a large number of incoming edges in the static call graph). We have performed a variety of studies on the proposed analysis. The experimental results show that, within the same time budget, the precision of the analysis is higher than that of a state-of-the-art highly-precise points-to analysis. In addition, the use of method summaries can lead to significant improvements in analysis performance.

Categories and Subject Descriptors

F.3.2 [**Logics and Meaning of Programs**]: Semantics of Programming Languages—*Program analysis*

General Terms

Algorithms, measurement, experimentation

Keywords

Alias analysis, context sensitivity, demand-driven

1. INTRODUCTION

Alias analysis is an essential component of virtually any tool for static analysis and transformation. It is especially important for analyzing modern object-oriented software systems, where pointers are used extensively to access heap objects. A may-alias analysis conservatively approximates, for two variables, whether they could point to the same object at run time. It is well-known that both the quality of such information and the cost of computing it can have significant influence on the effectiveness of software tools that make use of it. *Demand-driven* alias analyses have recently been the subject of several studies [3, 2, 27, 26, 33]. They are based on the observation that for most software tools, instead of requiring a whole-program points-to/alias solution, the tools are interested only in a small portion of variables and objects that are critical to the tasks being performed. A demand-driven analysis allows a client to perform *queries*, which can be answered efficiently, on demand, rather than computing a whole-program solution.

A typical way to answer queries regarding whether two variables are aliases is to query an underlying (demand-driven) points-to analysis, and then check whether their points-to sets have a non-empty intersection. Among the many points-to analyses for Java proposed in the literature, the analysis via context-free-language (CFL) reachability by Sridharan and Bodik [26] (*SB analysis* in the rest of the paper) is the most precise demand-driven one. However, it does not scale well when highly-precise results for a large number of queries are needed. Recently, a CFL-reachability-based demand-driven alias analysis [33] was proposed for the C language. This analysis is based on the insight that querying the aliasing relationship between two variables does not have to be performed by first computing points-to sets for them. Such a query can be answered much more efficiently by finding CFL-reachability paths on a graph based on a new context-free language.

While this formulation sheds new lights on the alias analysis implementation, it is not easy to adapt to Java. First, their context-free language considers only C-style pointers, where pointer dereferences can occur only through the dereference (*) operator. However, in Java, pointer dereferences can involve different fields, and a Java field-sensitive analysis has to match explicitly the fields associated with different pointer dereferences (i.e., at loads and stores of fields of heap objects). Second, this analysis is context-insensitive and does not distinguish calling contexts when determining aliasing relationships. This limits significantly its usefulness if it is ported directly to Java, as a Java program can make use of a large number of object-oriented data structures, and failing to distinguish calling contexts can lead to unusable solutions due to imprecision. As pointed out by numerous researchers (e.g., [4, 24, 13, 10]), context-sensitivity is one of the most important factors to be considered when designing a points-to/alias analysis for object-oriented programs. It remains a challenge to design a highly-precise demand-driven yet efficient alias analysis for Java, despite the high demand for such an analysis from modern software tools dealing with increasingly large and complex applications.

We propose a context-sensitive, demand-driven may alias analysis for Java, which does not require a points-to analysis to answer alias queries. The precision of the analysis is comparable to the state-of-the-art SB analysis [26] (a highly-precise demand-driven points-to analysis), as it context-sensitively models both pointer variables and pointer targets (i.e., heap objects). Yet, it can answer alias queries much faster than the traditional approach that uses points-to analysis. In fact, if they are run within the same time budget, our analysis often produces higher-quality solutions. We formulate the analysis as a single-source, single-target CFL-reachability problem. Similarly to the SB analysis, our analysis considers two dimensions: field accesses are checked to achieve field sensitivity, and method entries and exits are checked to achieve context sensitivity. Field sensitivity checks are based on a new context-free language *memAlias*, which is designed specifically for Java. This language has a much simpler structure than the language used in the CFL-reachability formulation from the SB analysis; as a result, we obtain a more efficient algorithm.

While our field-sensitivity check is more efficient than the one from the SB analysis, our analysis could still experience "context blowup". One typical approach for solving this problem is to introduce approximations, trading off precision for analysis running time (e.g., with approximation graph edges, as done in the SB analysis). Instead, our analysis attempts to solve the problem by computing *procedural reachability summaries* for methods that are invoked from many call sites. Summaries are computed *online*, during the analysis, and applied when CFL-reachability paths are about to enter these methods. This leads to significant reduction on the effort to re-traverse their bodies many times (potentially under different calling contexts). While trade-offs may still be employed later (e.g., when the client-defined budget for a query runs out), using such summaries can effectively avoid wasting the time budget on repeated work and thus postpone the out-of-budget timeout, leading to significant precision improvements.

The ability to compute and use reachability summaries is due to the special structure of the proposed *memAlias* language; it would be much more difficult to compute similar summaries for the SB analysis. Even though the *memAlias* (field-sensitivity) formulation is not as precise, it enables summary generation and yields an efficient algorithm. As a result, our analysis is able to traverse more contexts within the same budget. This results in a certain level of precision recovery, leading to even more precise solutions in many cases. The key novel observation is that certain loss of precision along the axis of field sensitivity can be successfully compensated by the increased precision along the axis of context sensitivity.

We have implemented the analysis on the Soot analysis framework [28] and performed a variety of studies on it. The results show that, given the same time budget for an alias query, the precision of our alias analysis in general is higher than that of the SB analysis, and in some cases significantly more precise aliasing relationships are reported. We have also investigated empirically the trade-offs for different choices of criteria for computing reachability summaries for methods, and observed the significant analysis improvements they could yield.

2. BACKGROUND

This section outlines the CFL-reachability formulation for a context-sensitive points-to analysis proposed in [26] by Sridharan and Bodik. A running example is also provided to illustrate the key ideas of our alias analysis.

CFL-reachability formulation of points-to analysis. In a directed graph G with labeled edges, a relation R over graph nodes can be formulated as a CFL-reachability problem by defining a context-free grammar and considering the corresponding language L. A pair $(n, n') \in R$ if and only if there exists a path from n to n' such that the sequence of edge labels belongs to L; such a path is an L-path. Node n' is L-reachable from n (denoted by $n\ L\ n'$), if there exists an L-path from n to n'. For any non-terminal S in the grammar, S-paths can be defined similarly.

This formulation has been used by existing points-to analyses for Java [26, 27] to model field sensitivity via object field reads/writes, and context sensitivity via method entries/exits. In this approach, a demand-driven points-to analysis determines all nodes n' such that $n\ L\ n'$ for a given node n, where language L is defined as $L_\mathsf{F} \cap L_\mathsf{C}$. Language L_F, where F stands for "flows-to", encodes the flow of reference values through object fields. Language L_C, where C stands for "calling-context", ensures a degree of calling context sensitivity.

Flow graph. To perform an analysis using the above formulation, a Java program is represented as a flow graph. In this graph G, if a run-time heap object represented by the abstract location o can flow to a variable v, there exists an L_F-path from o to v. There are four types of edges in the graph, created by considering the following categories of statements: an edge $o \xrightarrow{\mathsf{new}} x$ for an allocation $x = \text{new } O$; an edge $y \xrightarrow{\mathsf{assign}} x$ for an assignment $x = y$; edges $y \xrightarrow{\mathsf{store(f)}} x$ and $y \xrightarrow{\mathsf{load(f)}} x$ for a field write $x.f = y$ and a field read $x = y.f$, respectively. Parameter passing is treated as assignments from actuals to formals; return values are handled similarly. Accesses to arrays are represented by collapsing all array elements into one artificial field *arr_elm*.

Language L_F. If the flow graph G contains only new and assign edges, language L_F is regular and its grammar is simply *flowsTo* $\rightarrow$ $\mathsf{new}\,(\,\mathsf{assign}\,)^*$, showing the transitive flow due to assign edges. Clearly, o *flowsTo* v in G means that o belongs to the points-to set of v.

In the presence of field accesses, inverse edges are introduced to allow a CFL-reachability formulation. For each edge $x \xrightarrow{t} y$, an edge $y \xrightarrow{\bar{t}} x$ is introduced. For any path p, the inverse path $\bar{p}$ is defined by reversing the order of edges in p and replacing each edge with its inverse. Thus, if there exists a *flowsTo*-path from abstract object o to variable v, there also exists a $\overline{\textit{flowsTo}}$-path from v to o.

By definition, two variables a and b may alias if there exists an object o such that o can flow to both a and b. With inverse edges and inverse paths, the may-alias relation can be

```
1 class Vector {
2    Object[] data;
3    int cntr;
4    Vector() { t = new Object[MAX];
5            this.data = t; }
6    void add(Object e) {
7       t = this.data;
8       t[cntr++] = e;
9    }
10   Object get(int idx) {
11      t = this.data;
12      p = t[idx]; return p;
13   }
14 }
```

```
15 class Element { Object f; }
16 static void main(String[] args) {
17    Vector v = new Vector();
18    Element a = new Element();
19    t = "hello";  a.f = t;  v.add(a);
20    Vector w = new Vector();
21    Element b = new Element();
22    w.add(b);
23    Element c = (Element)w.get(0);
24    t = "world"; c.f = t;
25    Element d = (Element)v.get(0);
26    Object m = d.f;
27    System.out.println(m);
28 }
```

Figure 1: Running example.

modeled by defining a non-terminal *alias* such that $alias \rightarrow \overline{flowsTo}\, flowsTo$. The field-sensitive points-to relation can be defined by

$$flowsTo \rightarrow \mathsf{new}\,(\,\mathsf{assign} \mid \mathsf{store(f)}\ \ alias\ \mathsf{load(f)}\,)^*$$

This definition checks for balanced pairs of store(f) and load(f) operations, while also considering the potential aliasing between the base variables through which the field dereferences occur.

Language L_{C}***.*** Context sensitivity ensures that method entries and exits are properly matched. The standard balanced-parentheses formulation is

$$C \rightarrow \mathsf{entry(i)}\ C\ \mathsf{exit(i)} \mid C\ C \mid \epsilon$$

Here entry(i) and exit(i) correspond to the i-th static call site in the program. This production describes only paths that start and end in the same method, so it covers only a subset of the language. The full definition of L_{C} also allows a prefix with unbalanced closed parentheses and a suffix with unbalanced open parentheses [26]. Context sensitivity is achieved by considering entries and exits along an L_{F}-path and ensuring this sequence is in L_{C}. For the purposes of this context-sensitivity check, an $\overline{\mathsf{entry}}$ edge is treated as an exit edge and vice versa.

Running example. The code snippets in Figure 1 illustrate a simplified implementation of a Vector data structure and its client code. The corresponding graph representation for the Sridharan-Bodik CFL-reachability formulation is shown in Figure 2a.

Each object is represented as o_i, where i is the line number of the code where the object is created. For example, o_{19} represents the string literal “hello” at line 19. Similarly, each variable v is represented as v_i, where i denotes the line number of the code where v is declared. A special field *arr_elm* is used to represent any array element of an array. There is a *flowsTo*-path from o_{18} to d_{25}, because (1) there exist two field load/store edge sequences: $o_{18}\ new\ a_{18}\ store(arr\_elm)\ t_7$, and $t_{11}\ load(arr\_elm)\ p_{12}\ d_{25}$; (2) $t_7\ alias\ t_{11}$ (as o_4 can flow to both variables; this part of the graph is omitted for simplicity); and (3) the load and store edges in this path are properly balanced. In addition, this *flowsTo*-path is context-sensitive, as the sequence of method entry and exit edges is also balanced (i.e., $entry_{19}\ \overline{entry_{19}}\, entry_{25}\, exit_{25}$).

Note that for part (2) of the path, in order to verify that t_7 and t_{11} are aliases, the analysis needs to find two separate *flowsTo*-paths from the same object o_4 to them. In fact, this relationship can be seen much more easily without understanding which specific objects they point to: both are retrieved from the `data` field of object `this` in class `Vector`. Since there exists a `Vector` object on which both `add` and `get` are invoked, $\mathtt{this}_7$ and $\mathtt{this}_{11}$ can alias, and thus, t_7 and t_{11} are also aliases. Intuitively, we can design a new grammar that takes advantage of this (partial) knowledge to answer alias queries, leading to improved quality and efficiency. This may be especially useful in answering alias queries regarding variables that are close to each other (e.g., declared in the same method) while the objects that they point to are far away from them (e.g., across many method invocations).

As another example, a data dependence analysis needs to query whether or not variable pairs (a, d) and (c, d) are aliases, in order to construct define-use relationships (involving both stack and heap locations) for method `main`. For example, a dependence edge should be added between statements $m = d.f$ and $c.f = t$, if c and d can alias. Using this points-to analysis, answering this query requires the analysis to traverse the graph twice to find two long *flowsTo*-paths (i.e., from o_{21} to c_{23} and from o_{18} to d_{25}) and then decide that c and d cannot alias because their points-to sets do not contain a common object. Using our formulation for alias analysis, only one graph traversal is needed, based on the graph shown in Figure 2b.

3. DEMAND-DRIVEN ALIAS ANALYSIS

This section presents our context-sensitive demand-driven alias analysis. The section starts by discussing a program representation, *symbolic points-to graph*, which is used as the basis for the analysis. A may alias query performed on-demand is formulated as a single-source single-target CFL-reachability problem on this graph [18]. A context-insensitive version of the analysis is then described over a simplified language, and is later extended to a fully context-sensitive one by simultaneously performing CFL-reachability for a language that models matched method entries and exits.

3.1 Symbolic Points-to Graph (SPG)

The symbolic points-to graph was first proposed in our previous work [32] for an exhaustive whole-program alias analysis. It is adapted here for a different purpose. An SPG for a method is a *locally-resolved* points-to graph: it contains real points-to relations that can be resolved within the method while using placeholder *symbolic nodes* to represent objects that are created outside the method.

The SPG construction is done in two steps: in the first step, an intraprocedural SPG is built for each method; and in the second step, method-level SPGs are connected to form the interprocedural SPG (ISPG). Eventually, may alias

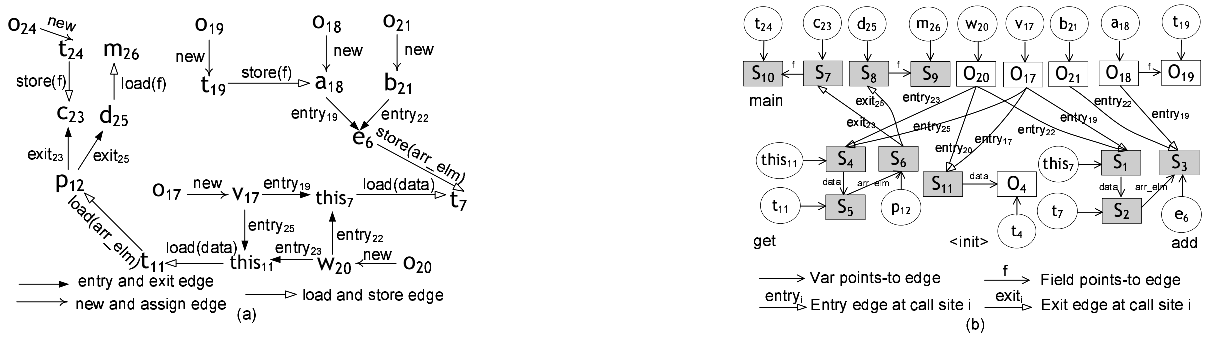

Figure 2: Program representations for the running example: (a) flow graph used by the Sridharan-Bodik formulation, and (b) interprocedural symbolic points-to graph (ISPG) used by our formulation.

queries are answered on-demand by performing graph traversals on the ISPG.

Each intraprocedural symbolic points-to graph has the following types of nodes and edges: (1) local variables $v \in \mathcal{V}$; (2) allocation nodes $o \in \mathcal{O}$; (3) symbolic nodes $s \in \mathcal{S}$ representing outside objects; (4) edges $v \to o_i \in \mathcal{V} \times \mathcal{O}$ representing that variable v may point to allocation node o_i; (5) edges $v \to s_i \in \mathcal{V} \times \mathcal{S}$ representing that variable v may point to an object defined outside of the current method, with the symbolic node s_i used as a placeholder for that object; (6) edges $o_i \xrightarrow{f} o_j \in (\mathcal{O} \cup \mathcal{S}) \times \mathit{Fields} \times (\mathcal{O} \cup \mathcal{S})$ representing that the field f of o_i may point to o_j.

There are multiple possible ways to construct an SPG. For ease of understanding, we present an SPG construction approach based on the Sridharan-Bodik (SB) CFL-reachability formulation discussed in Section 2. First, symbolic nodes are introduced, by modifying the flow graph described in Section 2 as follows. (1) For each formal parameter f of a method: edge $s \xrightarrow{\mathsf{new}} f$, where symbolic node s represents the object passed from outside; (2) For each call site $v = m()$: edge $s \xrightarrow{\mathsf{new}} v$, where symbolic node s is used to represent the return object of the call; (3) For each field access $v.f$ that has been read at least once in the method: edges $s \xrightarrow{\mathsf{new}} t$ and $t \xrightarrow{\mathsf{store}(f)} v$, where symbolic node s is used to represent the heap location that $v.f$ points to, and temporary variable t connects s and $v.f$.

The construction of the SPG is done by computing intraprocedural *flowsTo*-paths on this augmented flow graph. Given these paths, a points-to edge $v \to o$ is added to the SPG if o *flowsTo* v. A points-to edge $o_1 \xrightarrow{f} o_2$ is added to the SPG whenever there exists a flow graph edge $y \xrightarrow{\mathsf{store}(f)} x$ such that o_1 *flowsTo* x and o_2 *flowsTo* y.

To perform an interprocedural analysis, method-level SPGs are connected to build an interprocedural symbolic points-to graph (ISPG). The ISPG, simply a set of SPGs connected by edges representing calls, is still not a fully resolved points-to graph. For each call site of the form $r = a_0.m(a_1, ..., a_i, ...)$ and each formal-actual pair (f_i, a_i) of this call site, an *entry ISPG edge* is added from any allocation or symbolic node that a_i points to (in the SPG of the caller), to the symbolic node s created in the callee as a placeholder for f_i. For the return value, an artificial return variable *ret* is introduced in the callee, and an *exit ISPG edge* is added from any allocation or symbolic node that *ret* points to (in the SPG of the callee), to the symbolic node s which was introduced as a placeholder of the return value at the call site. These entry and exit edges are labeled with the call site's unique identifier, which will be used later to achieve context-sensitivity in the alias analysis.

The ISPG for the running example is shown in Figure 2b. Circles, white boxes, and shaded boxes represent variables, allocation sites, and symbolic nodes, respectively. The ISPG is a partially-resolved heap graph and relationships among symbolic nodes are unclear at this time—for example, it is possible for different symbolic nodes to represent the same object. The goal of the alias analysis is to make these relationships clear by computing CFL-reachability on this graph.

In this analysis, the aliasing relation between two variables is defined in terms of the *memory aliasing* between the allocation or symbolic nodes they point to. In other words, for two variables v_1 and v_2 in $\mathcal{V}$, if there exist pointed-to nodes o_1 and o_2 in $\mathcal{O} \cup \mathcal{S}$ such that there is a *memAlias*-path between o_1 and o_2 in the ISPG, then v_1 and v_2 are aliases. (The *memAlias* context-free language will be defined shortly.) Since *memAlias* is computed only along paths containing allocation and symbolic nodes, the discussion below will use "nodes" to refer to such nodes only, unless noted otherwise.

3.2 Context-Insensitive Alias Analysis

Two nodes o_1 and o_2 are memory aliases if they can represent the same run-time object. The memory alias relation ($\subseteq (\mathcal{O} \cup \mathcal{S}) \times (\mathcal{O} \cup \mathcal{S})$) can be described using a context-free language *memAlias* defined as follows:

$$\begin{aligned} \mathit{memAlias} \;&\to \bar{f}\ \mathit{memAlias}\ f \mid \mathit{memAlias}\ \ \mathit{memAlias} \\ &\mid \mathit{entry} \mid \overline{\mathit{entry}} \mid \mathit{exit} \mid \overline{\mathit{exit}} \mid \epsilon \end{aligned}$$

Informally, two nodes o_1 and o_2 are considered to be memory aliases if (1) they are reachable from nodes o_3 and o_4, (2) the field label strings on the two paths are the same, and (3) o_3 *memAlias* o_4. For now, context sensitivity is not considered and thus ISPG entry and exit edges are not matched; the context sensitivity checks will be discussed shortly. As an example, using this formulation we can quickly determine that t_7 and t_{11} are aliases (in Figure 2b), because symbolic nodes s_2 and s_5 (that t_7 and t_{11} point to, respectively) are reachable from node o_{20} and the two field edge label sequences are both "data". Note that the identification of this relationship does not require any knowledge of what are the actual objects that s_2 and s_5 can represent.

***memAlias* vs. *alias* in *[26]*.** It is important to note that our *memAlias* formulation simplifies the alias analysis by using *heap reachability* (i.e., sequences of points-to edges)

Algorithm 1: Context-sensitive alias analysis.

```
   Input: symbolic/alloc node $n_1$, symbolic/alloc node $n_2$
   Output: true/false indicating whether $n_1$ and $n_2$ can be
           memory aliases
1  if $n_1 = n_2$ then
2      return true
3  $worklist \leftarrow \{(n_1, \emptyset, \emptyset)\}$   // worklist of (node, stack, stack)
4  while $worklist \neq \emptyset$ do
5      remove a triple $(n, fldStk_n, cxtStk_n)$ from $worklist$
6      foreach $edge\ e \in incomingEdges(n)$ do
7          $(fldStk,\ cxtStk) \leftarrow clone(fldStk_n,\ cxtStk_n)$
8          if $e$ is a $FieldPointsToEdge \xrightarrow{f}$ then
9              $fldStk.push(f)$
10         else if $e$ is an $EntryEdge\ entry_i$ then
11             if $cxtStk \neq \emptyset$ then
12                 if $cxtStk.top() = i$ then
13                     $cxtStk.pop()$
14                 else
15                     continue
16         else if $e$ is an $ExitEdge\ exit_i$ then
17             $cxtStk.push(i)$
18         if $source(e) = n_2$ then // check for termination
19             if $fldStk = \emptyset$ then
20                 return true
21             else
22                 continue
23         $worklist \leftarrow worklist \cup \{(source(e), fldStk, cxtStk)\}$
24     foreach $edge\ e \in outgoingEdges(n)$ do
25         $(fldStk,\ cxtStk) \leftarrow clone(fldStk_n,\ cxtStk_n)$
26         if $e$ is a $FieldPointsToEdge \xrightarrow{f}$ then
27             if $fldStk.top() = f$ then
28                 $fldStk.pop()$
29             else
30                 continue
31         else // entry and exit edges handled in a similar way
32             ...
33         if $target(e) = n_2$ then // check for termination
34             if $fldStk = \emptyset$ then
35                 return true
36             else
37                 continue
38         $worklist \leftarrow worklist \cup \{(target(e), fldStk, cxtStk)\}$
39 return false
```

to approximate the actual *pointer value flows* (i.e., sequences of assignments) modeled by language *alias* (defined in Section 2) in the SB formulation. This simplification may lead to spurious aliasing relationships. Despite the potential imprecision in handling field-sensitivity, the overall precision of our alias analysis has been shown to be even higher than the precision of the SB analysis within the same time budget. The simple structure (and especially the symmetry) of the language yields an efficient algorithm that has lower cost. This allows the analysis to spend a more significant portion of the time budget on the handling of context sensitivity (compared to the SB analysis), leading to a closer approximation of a fully context-sensitive solution.

3.3 Handling of Context Sensitivity

A context-sensitivity check can be performed along with the field-sensitivity check to ensure that entries and exits are properly matched. The grammar for this dimension of checking is defined in an expected way as follows:

$$C \rightarrow (_i\ C\ )_i \mid C\ C \mid f \mid \bar{f} \mid \epsilon$$
$$(_i \rightarrow entry(i) \mid \overline{exit(i)} \qquad)_i \rightarrow exit(i) \mid \overline{entry(i)}$$

In the absence of recursive calls, the total number of open parentheses in the graph is finite, and thus L_{C} is a finite regular language. The handling of recursion will be discussed shortly. In the analysis implementation, L_{C} is augmented to model partially matched parentheses, so that a valid L_{C}-path could contain unbalanced closing parentheses as a prefix and unbalanced opening parentheses as a suffix. Similarly to the handling of context-sensitivity in existing analyses, we design a trade-off framework that allows a programmer to specify a budget (i.e., either running time, or number of graph nodes processed) to answer each query. The analysis gives up a context-sensitive graph traversal and uses a context-insensitive solution if the user-defined budget runs out. An interesting observation from our experiments is that once the budget exceeds a certain value, the precision of the analysis remains almost constant. This is because variables that are true (context-sensitive) aliases are usually close to each other and a *memAlias*-path between nodes that they point to can be easily found within the budget.

A brief description of the analysis algorithm is given in Algorithm 1. A worklist is maintained to process graph edges that need to be traversed. Each worklist element is a triple that contains a node n, a stack *fldStk* that is used to solve *memAlias* reachability, and a stack *cxtStk* that models calling contexts. Node n is reachable from the starting node n_1 along a path that is a valid prefix of a context-sensitive *memAlias*-path. Inverse edges are not explicitly represented in the graph; traversing backward an incoming edge with label t (lines 6–23) is the equivalent of traversing the inverse edge with label $\bar{t}$. Stack *fldStk* describes a sequence of unmatched $\bar{f}$ labels for the fields f encountered along the path from n_1 to the current node n. Following an outgoing edge with a field label (line 26) requires matching this label with the last unmatched $\bar{f}$ which currently resides at the top of *fldStk* (line 27). If the target node n_2 is reached and *fldStk* is empty (at lines 19 and 34), the field edge labels along the path from n_1 to n_2 are properly matched.

Stack *cxtStk* describes a sequence of unmatched *entry* and $\overline{exit}$ labels seen on the path from n_1 to n. For example, at line 17, the call site i is remembered because an incoming exit edge was traversed backward, which corresponds to observing a label $\overline{exit(i)}$. The labels on the stack represent unmatched open parentheses $(_i$. When a closing parenthesis $)_i$ is observed (e.g., at line 10, where an entry edge is traversed backward), a context-sensitivity check against the top of the stack is performed if the stack is not empty. If it is empty, the unchanged stack is added to the worklist because the grammar allows unbalanced $)_i$ parentheses as a prefix of the L_{C}-path. The termination checks (lines 18–22 and 33–37) do not consider *cxtStk* since the grammar allows unbalanced $(_i$ parentheses as a suffix.

3.4 Handling of Recursive Data Structures

One important technical challenge is the handling of recursion, as a typical context-sensitive analysis does not terminate in the presence of recursion without appropriate approximations. One such approximation, as used in [29], is to treat methods in a strongly connected component (SCC) of the call graph in a context-insensitive manner. However, significant precision loss could result from this handling [10], since an SCC in a (pre-computed) context-insensitive call

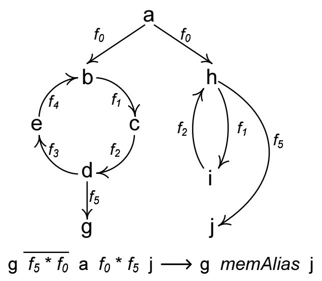
a
f_0
f_0
b
h
f_4
f_1
e
c
f_2
f_1
f_5
f_3
f_2
i
d
f_5
g
j
g $\overline{f_5 * f_0}$ a $f_0 * f_5$ j $\longrightarrow$ g *memAlias* j

Figure 3: Handling of recursive data structures.

graph can contain thousands of methods due to spurious call graph edges. The SB analysis improves this approach by looking for CFL paths on *context-sensitive* call chains: the points-to set of a call site receiver variable is determined under the calling context for which this call site is encountered in the analysis. While this handling can improve precision to a certain degree (e.g., by reducing the size of an SCC), its effectiveness is still limited by the collapsing of SCCs. Intuitively, methods in an SCC should be modeled context-sensitively if they do not cause cycles in the data flow (e.g., *flowsTo*- and *alias*-paths) being computed, which determines the termination of the analysis.

We propose a different technique that maintains higher precision for SCC methods, by approximating *recursive data structures*, instead of *recursive calls*. This ability of the analysis is due to the special modeling of the heap structure in a symbolic points-to graph. Recursive calls do not have any influence on the termination of the analysis algorithm, unless recursive data structures are encountered during the traversal of such calls. In Algorithm 1, we maintain knowledge of each entry and exit edge that was already visited, and such a visited edge cannot be used to extend the current path. This handling is sound because (1) re-traversing a method can be safely avoided if it does not involve any data flow (e.g., always takes entry and exit edges); and (2) if data flow (i.e., through field edges) is involved in the traversal, a recursive data structure must have already been detected before following an entry or exit edge the second time. Hence, it is sufficient to focus only on the handling of recursive data structures, without consideration for recursive calls.

To approximate aliasing relationships in the presence of recursive data structures, cycles on a *memAlias*-path are detected *on-the-fly* along with the CFL-reachability computation. Field edge labels in each cycle detected are represented by a wildcard symbol (i.e., *), which can match an arbitrary sequence of (normal and inverse) field edge labels. Figure 3 illustrates our handling of recursive data structures. Note that each of the two cycles $\overline{f_2 f_1 f_4 f_3}$ and $f_1 f_2$ is represented by a * symbol. As a wildcard can represent arbitrary fields, there exists a *memAlias*-path from g to j. In the actual implementation of the analysis, labels that a specific wildcard * can represent are remembered, so that when this * is on the top of the stack, the analysis can quickly decide whether to match an edge with this *, or to pop the * and then use the next edge on the stack for matching.

For the context-insensitive *memAlias*-reachability computation, a cycle can be easily detected by checking whether an edge (to be processed) is already in stack *fldStk*. It is more difficult to detect a cycle when the checks of context-sensitivity and field-sensitivity are performed simultaneously (i.e, both languages *memAlias* and L_C are considered). For

```
Algorithm 2: Context-sensitive handling of recursive
data structures.
  Input: symbolic/alloc node $n_1$, symbolic/alloc node $n_2$
  Output: true/false indicating whether $n_1$ and $n_2$ can be
          memory aliases
1  if $n_1 = n_2$ then
2      return true
3  $worklist \leftarrow \{(n_1, \emptyset, \emptyset)\}$   // worklist of (node, stack, stack)
4  while $worklist \neq \emptyset$ do
5      remove a triple $(n, fldStk_n, cxtStk_n)$ from $worklist$
6      foreach $edge\ e \in incomingEdges(n)$ do
7          ...
8          $nodesInCycle \leftarrow \emptyset$
9          if $(i \leftarrow index(e, fldStk)) \geq$
           $0 \wedge match(fldStk.callingContextAt(i), cxtStk)$ then
10             for $j \leftarrow 0;\ j < i;\ j \leftarrow j + 1$ do
11                 edge $d = fldStk.pop()$
12                 $nodesInCycle \leftarrow$
                   $nodesInCycle \cup \{source(d)\} \cup \{target(d)\}$
13             $fldStk.push(*,\ cxtStk)$
14         foreach $node\ n' \in nodesInCycle$ do
15             foreach $edge\ e' \in$
               $incomingEdges(n') \wedge source(e') \notin nodesInCycle$ do
16                 if $e'$ is a $FieldPointsToEdge$ then
17                     $fldStk' \leftarrow clone(fldStk)$
18                     $fldStk'.push(e', cxtStk)$
19                     $worklist \leftarrow$
                       $worklist \cup \{(source(e'), fldStk', cxtStk)\}$
20                 else // $e'$ is an entry$_i$ or exit$_i$ edge
21                     ...
22     foreach $edge\ e \in outgoingEdges(n)$ do
                       // handled in a similar way
23 return false
```

example, an `ArrayList` can have another `ArrayList` as its element. In this case, a field edge from a symbolic node representing an `ArrayList` object in one of its methods (such as `add`) to a symbolic node representing its internal array may be visited multiple times during a single traversal. However, this does not indicate a recursive data structure, but instead, it is due to the nesting of different objects in the data structure. We solve this problem by augmenting each edge in *fldStk* with a context stack that represents the call chain under which this edge is processed. To detect a cycle, we check (1) whether a field edge to be processed is in *fldStk* and if it is (2) whether the current calling context (represented by stack *cxtStk*) matches the calling context associated with the existing field edge in *fldStk* (the latter should be a prefix of the former).

The detailed algorithm for handling of cycles is shown in Algorithm 2. Once a cycle is detected (i.e., an edge e appears twice and the contexts match each other, shown at line 9), edges in stack *fldStk* between the two occurrences of e are replaced with a *; this new element of *fldStk* is associated with the current calling context *cxtStk* (lines 10–13). Set *nodesInCycle* contains all nodes in this cycle. Eventually, nodes that are adjacent to the cycle but are not in the cycle are added to the worklist (lines 14–21). These nodes will be processed regardless of whether or not they have been visited before, because a new cycle is identified and this could potentially lead to the identification of new aliasing relationships.

4. COMPUTING PROCEDURAL REACHABILITY SUMMARY

A method can be analyzed multiple times (under different calling contexts) during the CFL-reachability computation. This is usually the case even for answering one single alias query. To see this, consider the example from the last section, where an `ArrayList` object o_1 is added into another `ArrayList` object o_2. If the aliasing relationship to be queried involves an object retrieved from o_1, the graph traversal would go through methods `add` and `get` at least twice in order to understand the data structure. While analyzing a method multiple times can degrade significantly the analysis performance, it is not straightforward to decide how to speed up this process, as the method can be reached from different calling contexts, under which its behaviors can be completely different.

Existing work such as the SB analysis attempts to cache the (intermediate) analysis result for a method under a certain context, so that this result can be reused when the method is re-analyzed under this particular context. However, the effectiveness of this approach is limited in the following two important aspects. First, results can be cached only for specific contexts, while a method can be analyzed under a great number of different contexts. Second, the analysis can enter and exit a method through different parameters (including the return value), and it is unclear how caching should be performed for these different entries and exits. Hence, caching is often done in an ad-hoc manner (i.e., there does not exist a systematic way to perform it). Caching of analysis results is usually performed within a single query and cannot work across queries.

It is well-known that the effort of reanalyzing methods can be effectively reduced by using *procedural summaries* (e.g., [31, 1, 30, 22, 21, 23]). However, it is unclear how to compute summaries for a demand-driven analysis based on a CFL-reachability formulation. In this section, we propose to compute a *procedural reachability summary* (PRS) for a method, which summarizes a set of partial *memAlias* paths that go through this method—that is, each path enters the method through an *entry* or $\overline{exit}$ edge and leaves the method through an $\overline{entry}$ or *exit* edge. This summary is computed *online* during the CFL-reachability computation and is applied every time this method is reached in the analysis.

More formally, for a method m, consider the set E of symbolic or allocation nodes, where each node has either an incoming $entry_i$ edge or an outgoing $exit_i$ edge (where i represents a call site that invokes m). A PRS for m is a set of paths, each of the form $n_1 \overset{p}{\hookrightarrow} n_2$, where $n_1, n_2 \in E$ and $n_1 \neq n_2$. Here p is either a sequence of field points-to edges $(f_1 f_2 f_3 \ldots f_n)$ or a sequence of inverse field points-to edges $(\overline{f_1}\ \overline{f_2}\ \overline{f_3}\ \ldots\ \overline{f_n})$, summarizing the reachability information between n_1 and n_2. Note that the PRS also contains $n_2 \overset{\bar{p}}{\hookrightarrow} n_1$. For example, the PRS for method `add` in Figure 1 contains the sequence $p = (data, arr\_elm)$ from node S_1 to node S_3, as well as the corresponding sequence $\bar{p}$ in the opposite direction.

The summary computation is formulated as an all-pairs CFL-reachability problem over the node set E. Because the goal is to find field edge strings between any two nodes n_1 and n_2 in E, the CFL-reachability computation is still performed as described earlier, but this time we allow unbalanced field edges. The actual graph paths traversed during the summary computation start at n_1 and end at n_2. Each such path can (1) cross m and all methods (directly or transitively) invoked by m, (2) have matched entry/exit edges, and (3) have both matched and unmatched field edges. The entry/exit edges along the path correspond to balanced parentheses for language L_{C}. To construct the summary string p for $n_1 \overset{p}{\hookrightarrow} n_2$, all matched (entry/exit or field) labels are excluded from the path, leaving only the unmatched field edges.

When a call site that invokes m needs to be analyzed later in a graph traversal—that is, when a node n_1 in the set E of m is reached through an entry or inverse exit edge—the summary paths $n_1 \overset{p}{\hookrightarrow} n_2$ are retrieved. To apply the summary, all field labels in sequence p are pushed onto (or popped from) *fldStk*; whether to perform pushes or pops is determined by the edge direction in p. Next, any node outside m and connected with n_2 by an exit or inverse entry edge (properly matching the entry/exit edge reaching n_1) is added into the worklist for further processing. At a very high level, this process is similar to the use of summary edges in a system dependence graph for interprocedural program slicing [19].

Note that a method's PRS encodes only its partial reachability information, that is, paths that cannot reach a node in E are omitted from the summary. It would be extremely expensive for a method summary to contain the complete reachability information (not only in the method but also in methods reachable from it). Thus, applying m's summary at a call site that invokes m can cause unsoundness if any node involved in an alias query is either in m itself or a method reachable from m in the call graph. To solve this problem, for each query regarding the (memory) aliasing relationship between nodes n_1 and n_2, we first check whether the methods that contain n_1 and n_2 are reachable from m, upon reaching a call site that invokes m. If they are, we perform regular CFL-reachability computation instead of applying summaries. If these methods are not reachable from m, m's PRS can be safely applied.

As PRSs are computed *online*, interleaved with the regular CFL-reachability analysis, a key factor for analysis performance is the selection of an appropriate set of methods for which summaries are computed. This is because the computation of a method's PRS (i.e., all-pairs CFL-reachability) is more expensive than a regular traversal of the method (i.e., single-source CFL-reachability). Hence, it would be useless, or even harmful, to compute summaries for methods that are invoked from a small number of call sites. Furthermore, methods whose nodes are likely to be frequently queried should not be considered for summary computation as their summaries are not used anyway when nodes in them are in a query. From our experience, *library methods* (e.g., methods in data structure classes `HashMap`, `ArrayList`, etc.) are good candidates for summary generation, as these methods can be called a very large number of times in real-world programs. In addition, a client analysis (that makes use of this alias analysis) usually focuses on the *application code* and thus is unlikely to make queries regarding aliasing relationships among nodes in these library methods.

We use a simple metric to choose such library methods. For each method m, we consider the ratio between the number of incoming call graph edges for m, and the average number of incoming call graph edges for all methods. When this value exceeds a certain threshold, m's summary is com-

Benchmark	#Methods	#Statements	#SB/ISPG Nodes	#SB/ISPG Edges	#Queries
compress	2344	43938	18778/10977	18374/3214	83421
db	2352	44187	19062/11138	18621/3219	62478
jack	2606	53375	22185/12605	21523/15560	160170
javac	3520	66971	23858/14119	23258/3939	386724
jess	2772	51021	22773/13421	21769/4754	257032
mpegaudio	2528	55166	22446/12774	21749/4538	222432
mtrt	2485	46969	20344/11878	19674/3453	120042
soot-c	4583	71406	31054/18863	29971/5010	712816
sablecc-j	8789	125538	44134/26512	42114/9365	357012
jflex	4008	25150	31331/18248	30301/4971	159046
muffin	4326	80370	33211/19659	32497/5282	1234635
jb	2393	43722	19179/11275	18881/3146	9045
jlex	2423	49100	21482/11787	20643/3846	51480
java_cup	2605	50315	22636/13214	21933/3438	14534
polyglot	2322	42620	18739/10950	18337/3128	1600116
antlr	2998	57197	25505/15068	24462/4116	528891
bloat	4994	79784	38002/23192	35861/5428	1326732
jython	4136	80067	34143/19969	33970/5179	971376
ps	5278	84540	39627/23601	38746/5646	5000

Table 1: Benchmark characteristics.

puted. While this is a simple metric, it is quite effective in selecting methods for which it is worth computing summaries. The next section presents experimental studies with different threshold values.

5. EVALUATION

The proposed alias analysis was implemented using the Soot 2.3.0 program analysis framework for Java [28]. In order to compare our analysis with previous work (such as [26]), the Sun JDK 1.3.1_01 library was used in all studies. Experiments are performed on a machine with an Intel Xeon 2.83GHz processor and a maximum JVM heap space of 2GB. A total of 19 Java programs were studied in our experiments, and their characteristics are shown in Table 1 (this information also appears in [32]). The first two columns "#Methods" and "#Statements" show the total numbers of methods in context-insensitive call graphs constructed by the Spark framework in Soot [9] and the numbers of statements in these methods. Shown in columns "#SB/ISPG Nodes" and "#SB/ISPG Edges" are the total numbers of nodes (edges) in the program representation used in the SB analysis [26] (i.e., the flow graph) and our representation (i.e., the ISPG restricted to allocation/symbolic nodes, since only they can occur on *memAlias*-paths). The numbers of nodes and edges in our representation are much smaller than those in the flow graph. The last column shows the total number of queries performed in each benchmark, and will be discussed shortly.

5.1 Methodology

The precision and performance of the proposed analysis are compared against those of a traditional alias analysis that uses points-to analyses to determine aliasing relationships. Here we consider two state-of-the-art points-to analyses, namely the SB points-to analysis in [26] and the 1-object-sensitive points-to analysis implemented in the Paddle context-sensitive analysis framework [14]. To simulate how a client analysis would use the alias analysis, we implemented a data dependence client that performs alias queries regarding variables x and y for all pairs of heap dereference statements that access the same field f (i.e., $x.f$ and $y.f$) where at least one statement writes to f. If x and y may alias, the two statements may access the same heap location and thus may have a data dependence. All possible pairs of heap dereference statements are considered so that the alias queries used for evaluation are not biased. Dependence analysis is an important component of many real-world software tools, such as program slicers and data race detectors.

An inexpensive context-insensitive points-to analysis (in Spark) is performed as a first step to prune pairs of statements that can be determined to access different heap locations in a context-insensitive setting. Next, a context-sensitive analysis (i.e., our analysis or the SB analysis) is performed to further prune the alias pairs. We found that the 1-object-sensitive analysis from [14] (which is a whole-program analysis) runs an order of magnitude slower than our alias analysis and the SB analysis (both of which are on-demand analyses), and yet its precision is lower. Hence, we choose not to include comparison details; instead, we focus on the comparison between the SB analysis and our analysis.

5.2 Comparison with the SB Analysis

In this set of experiments, a fixed amount of time is considered as the per-query budget. By default, the SB analysis uses the number of nodes traversed as the metric for the budget. We modify the analysis to use the absolute time, in order to enable a fair comparison. This is because the two program representations are not the same and thus the numbers of nodes in the two graphs are not compatible metrics.

If the alias analysis cannot answer a query within this time limit, a context-insensitive solution is used instead. The total number of queries performed for each benchmark is shown in the last column of Table 1. We have compared the precision of the two analyses under different time budgets: 1ms, 2ms, 5ms, and 10ms. The results are shown in Table 2. Column SDA reports the precision of our analysis when summary generation is enabled. The reader can ignore this column for now; details about this comparison can be found later in Section 5.3.

Our analysis reports smaller numbers of alias pairs for 12, 13, 14, and 14 programs, under time budgets 1ms, 2ms, 5ms, and 10ms, respectively. For some of these programs, such as `javac`, our analysis reports a significantly smaller number of alias pairs than the SB analysis. Note that for this comparison, we do not compute and use procedural reachability summaries. The analysis would produce even more precise alias information when summaries are used (details will be discussed shortly).

Benchmark	SB(1ms)	DA(1ms)	SB(2ms)	DA(2ms)	SB(5ms)	DA(5ms)	SB(10ms)	DA(10ms)	SDA(10ms)
compress	1193	1192	1193	1192	1193	1192	1193	1192	1191
db	1039	1039	1039	1039	1039	1039	1039	1039	1038
jack	48807	48802	48806	48802	48804	48801	48804	48801	48801
javac	105512	59547	105485	59547	105472	59549	105439	59498	59481
jess	4047	4048	4047	4044	4047	4044	4047	4046	4046
mpegaudio	2982	2899	2978	2899	2978	2899	2972	2899	2899
mtrt	1611	1602	1611	1603	1604	1598	1604	1598	1596
soot-c	10378	10319	10363	10314	10363	10305	10381	10305	10285
sablecc-j	11447	11461	11447	11461	11447	11461	11445	11461	11461
jflex	6080	6077	6079	6078	6079	6078	6079	6076	6075
muffin	8154	8156	8153	8156	8151	8156	8151	8156	8156
jb	902	905	902	904	900	904	900	905	905
jlex	17756	17794	17747	17796	17747	17792	17748	17796	17758
java_cup	2909	2910	2908	2910	2908	2910	2908	2910	2910
polyglot	2933	2933	2933	2933	2933	2933	2933	2933	2933
antlr	27963	21303	27774	21303	27668	21300	27837	21300	21299
bloat	288085	288062	288085	288063	288085	288060	288086	288055	287570
jython	51453	51449	OM*	51446	OM*	51445	OM*	51445	51443
ps	4657	4408	4657	4411	4657	4410	4657	4411	4413

Table 2: Comparison between our analysis and the SB points-to analysis. Columns SB and DA show are the numbers of alias pairs reported by the SB analysis and by our demand-driven alias analysis (DA). Column SDA shows the numbers of alias pairs reported by the summary-based analysis under configuration $T = 2$. The numbers in parentheses are the time budgets used to answer each alias query. OM* means the analysis runs out of memory.

For the programs where the SB analysis is more precise, the results of the two analyses are actually very close. To understand why our analysis performs surprisingly better in some cases, we carefully inspected the analysis solutions for these cases. Below we report a concrete case study for `javac` that shows a typical situation that prevents the SB analysis from generating highly-precise solutions, and how our analysis ameliorates the problem.

***Case study for* `javac`.** The SB analysis reports that statements `r0.left=$r12` in `ConditionalExpression.inline` and `r0.left=$r7` in `BinaryShiftExpression.selectType` can access the same heap location, while ours does not. By inspecting the program, we found that the points-to sets of the two `r0` variables computed by the SB analysis contain a common spurious object that obviously could not flow to either of these variables. The analysis starts with a context-insensitive points-to set and keeps refining it (i.e., following more and more precise *flowsTo*-paths until the budget runs out). This common spurious object is actually in the initial context-insensitive points-to sets of both `r0`. The SB analysis cannot filter it out within the budget primarily because there are very large SCCs in `javac`'s call graph (i.e., almost all methods are in SCCs). It is impossible to verify whether or not there exists a *flowsTo*-path by handling SCCs context-insensitively. In addition, the method that contains this spurious object is far away from methods `inline` and `selectType`, making it difficult to prove or disprove the existence of a CFL path. In fact, `inline` and `selectType` are very close to each other, and it is much easier for our analysis to check *memAlias*-paths between them. This is a common case that appeared often during our code inspection. We found similar frequently-occurring cases that could not be described due to space limitations. All of them may contribute to the precision difference between the two analyses.

5.3 Evaluation of Summary-Based Analyses

To demonstrate the effectiveness of procedural reachability summary, we compare the total running times of the two versions of our analysis, one with the summary generation and the other without. For this set of experiments, the total number of nodes traversed in each query is used as the per-query budget (1000 nodes was used in this experiment). As the total number of nodes traversed can be considered as an indication of precision, the goal of using a fixed number of nodes as budget is thus to compare the running times of our analysis under different configurations to achieve roughly the same precision (an actual precision comparison is presented shortly).

Running time. For the summary-based analysis, different threshold values T are used (i.e., 1, 2, 5, 10, 15, and 20) to determine whether or not a method should be subjected to summarization. As discussed in Section 4, for a method m we consider the ratio between the number of incoming call graph edges of m, and the average number of incoming call graph edges for all methods. When this ratio is greater than T, the analysis computes and uses a reachability summary for m. To understand the performance of the analysis under these different configurations, for each such configuration, we compute the geometric mean (GM) of the analysis running times over the 19 programs. In general, the summary-based analyses were faster than the non-summary one ($GM_{nosumm} = 10970$ ms). For the six summary-based configurations studied, $T = 2$ ($GM_{summ(2)} = 8127$ ms) and $T = 20$ ($GM_{summ(20)} = 9205$ ms) are the ones with the best and the worst performance. Running times for the other configurations are between $GM_{summ(2)}$ and $GM_{summ(20)}$. In our experiments, $T = 2$ seems to be an appropriate threshold value that achieves a balance: computing summary for either too many methods ($T = 1$) or too few methods ($T > 2$) is less beneficial. Compared with the non-summary version, the analysis with $T = 2$ achieves an average running time reduction of 23.8%. A detailed comparison of the performance improvements under several configurations is shown in Figure 4.

Precision. We have also compared the precision of a summary-based version of our analysis against that of a non-summary-based version and of the SB analysis under the same time budget. We run our summary-based analysis under the best configuration that we have observed ($T = 2$), and specify 10ms as per-query budget. We chose this longest

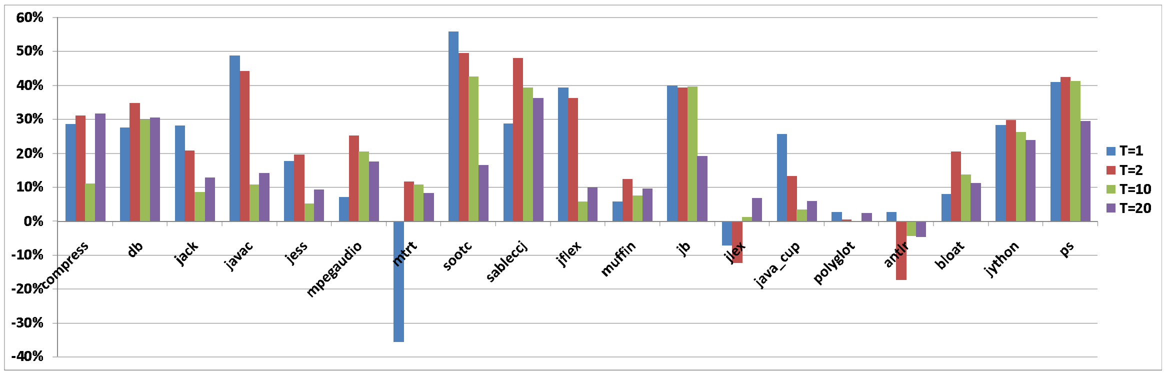

Figure 4: Comparisons among summary-based configurations. The y-axis shows the performance improvement over the non-summary version: $(RT_{nosumm} - RT_{summ})/RT_{nosumm}$ where RT stands for running time.

time (10ms) as the budget because the SB analysis achieves its highest precision under this time. The numbers of alias pairs reported by this summary-based analysis are shown in Column SDA of Table 2. For most programs in our benchmark set, SDA is more precise than both the non-summary-based analysis (column DA) and the SB analysis.

6. RELATED WORK

There exists a very large body of work on points-to/alias analysis. We refer the reader to [4, 24, 8, 25] for a broader discussion of these analyses. The description in this section is limited to the CFL-reachability-based analysis algorithms that are most closely related to our technique.

Work by Reps et al. [18, 19, 5, 16, 20] proposes to model realizable paths using a context-free language that treats method calls and returns as pairs of balanced parentheses. Sridharan et al. define a CFL-reachability formulation to precisely model heap accesses, which results in demand-driven points-to analyses for Java [27, 26]. Combining the CFL-reachability formulations of both heap accesses and interprocedural realizable paths, [26] proposes a context-sensitive analysis that achieves high precision by refining points-to relationships.

CFL-reachability can be used to implement a variety of static analyses, such as polymorphic flow analysis [15], shape analysis [17], and information flow analysis [11]. The work in [6, 12] studies the connection between CFL-reachability and set constraints, shows the similarity between the two problems, and provides new implementation strategies for problems that can be formulated in this manner. Work from [7] extends set constraints to express program analyses involving one context-free and any number of regular reachability properties.

Our previous work [32] proposes a whole-program alias analysis based on a CFL-reachability formulation that can be used to quickly terminate the traversal of a false *flowsTo* path in order to speed up the Sridharan-Bodik points-to analysis. This analysis does not provide on-demand capabilities, and achieves context-sensitivity by cloning SPG subgraphs. Zheng and Rugina [33] present a CFL-reachability formulation of alias analysis and implement a context-insensitive demand-driven analysis for C programs. The key insight is that aliasing information can be directly computed without having to compute points-to information first. While our analysis is based on this same insight, it is tailored for Java and is fully context-sensitive. This is accomplished by designing completely different context-free languages, formulations, and algorithms.

7. CONCLUSIONS

This paper presents a demand-driven alias analysis for the Java language. Unlike existing analyses for Java, it answers alias queries by computing CFL-reachability alias information directly, instead of relying on an underlying points-to analysis. The analysis performs field-sensitivity and context-sensitivity checks simultaneously over an interprocedural symbolic points-to graph—a heap structure approximation that contains locally-resolved points-to relationships and uses placeholder symbolic nodes to represent unidentified objects. This structure of the ISPG allows us to develop a simple context-free language *memAlias* and an efficient algorithm for context-sensitive CFL-reachability based on this language. To further improve analysis performance, we propose to compute and use online summaries for methods whose processing could have substantial effects on analysis running time. We have empirically compared the proposed approach with a state-of-the-art demand-driven points-to analysis [26]. We found that our analysis can produce more precise solutions than this existing analysis when they are run with the same time budget. The experimental results also show that when the set of methods for which summaries are computed is selected appropriately, the use of summaries can reduce significantly the analysis running time. These findings suggest that our analysis is a good candidate for use in software tools.

Acknowledgments

We thank the anonymous reviewers for their valuable and thorough comments. This material is based upon work supported by the National Science Foundation under CAREER grant CCF-0546040, grant CCF-1017204, and by an IBM Software Quality Innovation Faculty Award. Guoqing Xu was supported in part by an IBM Ph.D. Fellowship Award.

8. REFERENCES

[1] R. Chatterjee, B. G. Ryder, and W. Landi. Relevant context inference. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 133–146, 1999.

[2] S. Guyer and C. Lin. Client-driven pointer analysis. In *Static Analysis Symposium*, pages 214–236, 2003.

[3] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 24–34, 2001.

[4] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61, 2001.

[5] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 104–115, 1995.

[6] J. Kodumal and A. Aiken. The set constraint/CFL reachability connection in practice. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 207–218, 2004.

[7] J. Kodumal and A. Aiken. Regularly annotated set constraints. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 331–341, 2007.

[8] O. Lhoták. *Program Analysis using Binary Decision Diagrams*. PhD thesis, McGill University, 2006.

[9] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction*, pages 153–169, 2003.

[10] O. Lhoták and L. Hendren. Context-sensitive points-to analysis: Is it worth it? In *International Conference on Compiler Construction*, pages 47–64, 2006.

[11] Y. Liu and A. Milanova. Static analysis for inference of explicit information flow. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 50–56, 2008.

[12] D. Melski and T. Reps. Interconvertibility of a class of set constraints and context-free-language reachability. *Theoretical Computer Science*, 248:29–98, 2000.

[13] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, 2005.

[14] Paddle Framework, www.sable.mcgill.ca/paddle.

[15] J. Rehof and M. Fähndrich. Type-based flow analysis: From polymorphic subtyping to CFL-reachability. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 54–66, 2001.

[16] T. Reps. Solving demand versions of interprocedural analysis problems. In *International Conference on Compiler Construction*, pages 389–403, 1994.

[17] T. Reps. Shape analysis as a generalized path problem. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 1–11, 1995.

[18] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, 1998.

[19] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[20] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 11–20, 1994.

[21] A. Rountev, S. Kagan, and T. Marlowe. Interprocedural dataflow analysis in the presence of large libraries. In *International Conference on Compiler Construction*, pages 2–16, 2006.

[22] A. Rountev and B. G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *International Conference on Compiler Construction*, pages 20–36, 2001.

[23] A. Rountev, M. Sharp, and G. Xu. IDE dataflow analysis in the presence of large object-oriented libraries. In *International Conference on Compiler Construction*, pages 53–68, 2008.

[24] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *International Conference on Compiler Construction*, pages 126–137, 2003.

[25] Y. Smaragdakis, M. Bravenboer, and O. Lhotak. Pick your contexts well: understanding object-sensitivity. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 17–30, 2011.

[26] M. Sridharan and R. Bodik. Refinement-based context-sensitive points-to analysis for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 387–400, 2006.

[27] M. Sridharan, D. Gopan, L. Shan, and R. Bodik. Demand-driven points-to analysis for Java. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 59–76, 2005.

[28] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *International Conference on Compiler Construction*, pages 18–34, 2000.

[29] J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 131–144, 2004.

[30] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 187–206, 1999.

[31] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 1995.

[32] G. Xu, A. Rountev, and M. Sridharan. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *European Conference on Object-Oriented Programming*, pages 98–122, 2009.

[33] X. Zheng and R. Rugina. Demand-driven alias analysis for C. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 197–208, 2008.