On-Demand Dynamic Summary-based Points-to Analysis

Lei Shang
School of Computer Science & Engineering
The University of New South Wales, NSW, Australia
shangl@cse.unsw.edu.au

Xinwei Xie
School of Computer Science & Engineering
The University of New South Wales, NSW, Australia
xinweix@cse.unsw.edu.au

Jingling Xue
School of Computer Science & Engineering
The University of New South Wales, NSW, Australia
jingling@cse.unsw.edu.au

ABSTRACT

Static analyses can be typically accelerated by reducing redundancies. Modern demand-driven points-to or alias analysis techniques rest on the foundation of Context-Free Language (CFL) reachability. These techniques achieve high precision efficiently for a small number of queries raised in small programs but may still be too slow in answering many queries for large programs in a context-sensitive manner.

We present an approach, called DYNSUM, to perform context-sensitive demand-driven points-to analysis fully on-demand by means of computing CFL-reachability summaries without any precision loss. The novelty lies in initially performing a *Partial Points-To Analysis* (PPTA) within a method, which is field-sensitive but context-independent, to summarize its local points-to relations encountered during a query and reusing this information later in the same or different calling contexts. We have compared DYNSUM with REFINEPTS, a refinement-based analysis, using three clients (safe casting, null dereferencing and factory methods) for a suite of nine Java programs. DYNSUM's average speedups are 1.95×, 2.28× and 1.37×, respectively. We have also compared DYNSUM with a static approach, which is referred to STASUM here, to show its improved scalability for the same three clients.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Optimization*

General Terms

Algorithms, Languages, Experimentation, Performance

Keywords

Dynamic summary, points-to analysis, demand-driven analysis, CFL reachability

1. INTRODUCTION

Many static analyses can be accelerated if some redundant computations can be avoided. Considerable progress has been made, resulting in, for example, cycle elimination [4, 5] for Andersen-style points-to analysis [1] and sparse analysis [6, 7, 17, 24] for flow-sensitive points-to analysis. In the case of context-sensitive points-to analysis, computing a points-to summary for a method [13, 19, 24] avoids re-summarizing it unnecessarily for the same and different calling contexts. Despite a lot of earlier efforts, it remains unclear how to craft points-to analyses that can efficiently answer demand queries (e.g., non-aliasing) for a specific client.

The majority of the current solutions perform a whole-program points-to analysis to improve precision at the expense of efficiency, by computing points-to information for all variables in the program. Such exhaustive algorithms are too resource-intensive to be useful in environments with small time budgets, such as just-in-time (JIT) compilers and IDEs. One widely acceptable observation is that points-to analysis is not a stand-alone task since it needs to be tailored to suit the specific needs of a client application. As a result, much recent work [15, 16, 20, 25] has focussed on *demand-driven* points-to analysis, which mostly relies on Context-Free Language (CFL) reachability [14] to perform only the necessary work for a set of variables specified by a client rather than a whole-program analysis to find all its points-to information.

To perform points-to analysis with CFL reachability, a program is represented as a directed graph, with nodes denoting variables/objects and edges pointer-manipulating statements. Determining if a variable v points to an object o requires finding a path p between the nodes v and o in the graph such that p's label is in a CFL that ensures the corresponding statements can cause v to point to o. To balance precision and efficiency for on-demand queries, a points-to analysis is typically flow-insensitive, field-sensitive and context-sensitive [15]. Context-sensitivity is realized as a balanced-parentheses problem along two axes: method invocation (by matching call entries and exits so that only realizable paths are considered) and heap abstraction (by distinguishing the same abstract object from different paths).

While such CFL-reachability formulation is promising, performing demand-driven points-to analysis for large, complex software can still be costly, especially when a client issues a large number of queries. Existing solutions have addressed the performance issue from several directions, by using refinement [15], (whole-program) pre-analysis [20] and ad hoc caching [15, 20, 25]. However, redundant traversals along the same path are still repeatedly made, unless they are identified by a time-consuming whole-program pre-analysis. Among all these existing efforts, REFINEPTS [15] represents a state-of-the-art solution. However, its refinement approach is well-suited only to clients that can be satisfied early enough when most of heap accesses are still analyzed in a field-based manner rather than field-sensitively. Otherwise, its field-based refinement efforts (in terms of match edges) are pure overhead.

In this paper, we introduce a novel technique, called DynSum, to perform context-sensitive demand-driven points-to analysis fully on-demand. Unlike existing techniques [15, 20, 25], our approach exploits local reachability reuse by performing a *Partial Points-To Analysis* (PPTA) within a method dynamically. PPTA is field-sensitive but context-independent, thereby enabling the summarized points-to relations in a method to be reused in its different calling contexts without any precision loss. We identify such reuse as a practical basis for developing an effective optimization for demand-driven points-to analysis.

This paper makes the following contributions:

- We present a PPTA-based approach to boost the performance of CFL-reachability-based context-sensitive demand-driven points-to analysis by exploiting local reachability reuse. Our dynamic approach improves the performance of demand-driven points-to analysis without sacrificing precision and is *fully on-demand* without requiring any (costly) whole-program pre-analysis. This appears to be the first points-to analysis that computes dynamic method summaries to answer demand queries.
- We have implemented DynSum in the Soot compiler framework for Java. We have used three representative clients (safe casting, null dereferencing and factory methods) to evaluate the performance improvements against RefinePTS, the state-of-the-art demand-driven points-to analysis introduced in [15]. The average speedups achieved by DynSum for the three clients over a suite of nine Java benchmarks are 1.95×, 2.28× and 1.37×, respectively.
- We show that DynSum computes only a small percentage of the summaries computed by StaSum, a static whole-program analysis [22]. This makes DynSum more scalable and better-suited for answering demand queries in environments such as JIT compilers and IDEs, particularly when the program constantly undergoes a lot of edits.

2. PROGRAM REPRESENTATION

We consider Java programs although our approach applies equally well to C programs. Since the analysis is flow-insensitive, control-flow statements are irrelevant. By convention, parameter passing and method returns have assignment semantics. Local and global variables will be distinguished as global variables are context-insensitive. Therefore, in this paper, a program is represented with the syntax given in Figure 1.

A Java program is represented by a *directed graph*, known as a *Pointer Assignment Graph* (PAG), which has threes types of nodes, V, G and O. All edges are oriented in the direction of value flow, representing the statements in the program. A method m is associated with the following seven types of edges:

- new, $n_2 \xleftarrow{\textsf{new}} n_1$: n_1 is an object created and n_2 is a local variable both in method m, with ← indicating the flow of n_1 into n_2. As a result, n_2 points directly to n_1.

Integer (Call Sites)	i	$\in$	$\mathbb{N}$
Globals Variables	g	$\in$	G
Local Variables	v	$\in$	V
Fields	f	$\in$	F
Objects	o	$\in$	O
Allocations	$v = \textsf{new}\ o$	$\in$	$V \times O$
Assignments	$v_1 = v_2$	$\in$	$V \cup G \times V \cup G$
Loads	$v_1 = v_2.f$	$\in$	$V \times V \times F$
Stores	$v_1.f = v_2$	$\in$	$V \times F \times V$
Parameter Passing	$param \xleftarrow{\textsf{entry}_i} v$	$\in$	$V \times \mathbb{N} \mapsto V$
Returns	$v \xleftarrow{\textsf{exit}_i} ret$	$\in$	$V \times \mathbb{N} \mapsto V$

Figure 1: An abstraction of Java programs.

- assign, $n_2 \xleftarrow{\textsf{assign}} n_1$: n_1 and n_2 are local variables in method m. So n_2 points to whatever n_1 points to. Such edges represent *local assignments* in method m.
- assignglobal, $n_2 \xleftarrow{\textsf{assignglobal}} n_1$: n_1 or n_2 or both are static variables in a class of the program. So n_2 points to whatever n_1 points to. Such edges represent (context-insensitive) *global assignments* in the program.
- load(f), $n_2 \xleftarrow{\textsf{load}(f)} n_1$: n_1 and n_2 are local variables in method m and f is an instance field, with the statement representing the load $n_2 = n_1.f$.
- store(f), $n_2 \xleftarrow{\textsf{store}(f)} n_1$: n_1 and n_2 are local variables in method m and f is an instance field, with the statement representing the store $n_2.f = n_1$.
- entry$_i$, $n_2 \xleftarrow{\textsf{entry}_i} n_1$: n_1 is a local variable in a calling method that contains a call site at line i to method m, such that n_1 represents an actual parameter of the call and n_2 is its corresponding formal parameter of method m. So n_2 points to whatever n_1 points to.
- exit$_i$, $n_2 \xleftarrow{\textsf{exit}_i} n_1$: n_1 is a local variable that contains a return value of method m and n_2 is a local variable that is assigned from n_1 at a call site i in a calling method. So n_2 points to whatever n_1 points to.

Loads and stores to array elements are modeled by collapsing all elements into a special field arr. As is customary, it is assumed that no two classes (methods) contain the same identically named global (local) variable.

Figure 2 gives an example and its PAG representation. To avoid cluttering, the labels "assign" and "assignglobal" for assignment edges are omitted. Note that o_i denotes the object created at the allocation site in line i and v_{m} (with a subscript) denotes variable v declared in method m.

In the PAG shown, the edges are classified into *local edges* (new, assign, load and store) and *global edges* (assignglobal, entry$_i$ and exit$_i$). The local edges are enclosed inside dotted rectangles and the global edges span across them. DynSum aims to exploit the local reachability reuse across the local edges to accelerate its performance in answering demand queries.

```
1 class Vector{
2   Object[] elems;
3   int count;
4   Vector(){
5     t=new Object[8];
6     this.elems=t;}
7   void add(Object p){
8     t=this.elems;
9     t[count++]=p;}
10  Object get(int i){
11    t=this.elems;
12    return t[i]; }}
13 class Client{
14   Vector vec;
15   Client() {}
16   Client(Vector v)
17   { this.vec=v; }
18   void set(Vector v)
19   { this.vec=v; }
20   Object retrieve()
21   { t=this.vec;
22     return t.get(0); }}
23 class Main{
24 static void main(...){
25   Vector v1=new Vector();
26   v1.add(new Integer(1));
27   Client c1=new Client(v1);
28   Vector v2=new Vector();
29   v2.add(new String());
30   Client c2=new Client();
31   c2.set(v2);
32   s1=c1.retrieve();
33   s2=c2.retrieve();}
34 }
```

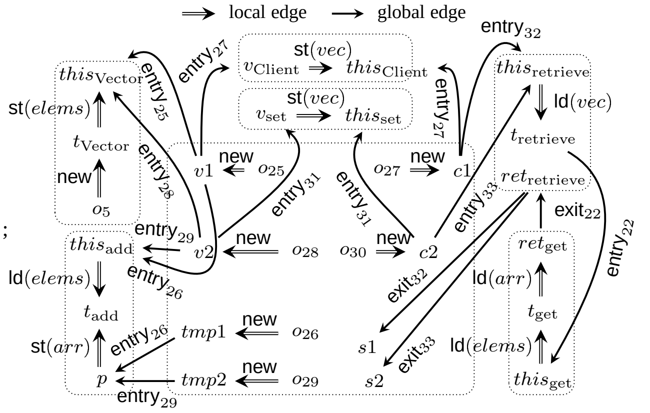
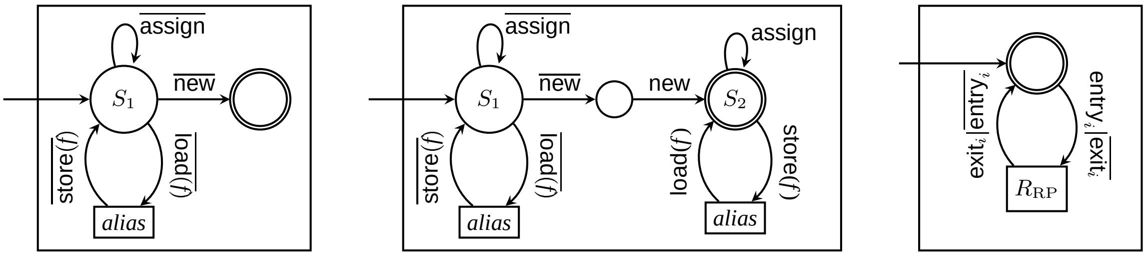


Figure 2: A Java example and its PAG.



(a) L_{FT} (with *pointsTo*, i.e., $\overline{\textit{flowsTo}}$ shown on the left and *alias* on the right)

(b) R_{RP}

Figure 3: Recursive State Machines (RSMs) for L_{FT} and R_{RP}.

3. BACKGROUND AND MOTIVATION

In this section, we introduce the state-of-the-art demand-driven points-to analyses for Java formulated in terms of CFL reachability by Sridharan and Bodík [15, 16]. These analyses provide the baseline for us to enhance its performance through dynamic reachability reuse.

Section 3.1 reviews CFL reachability. Section 3.2 describes the CFL reachability formulation of a field-sensitive points-to analysis presented in [16]. This earlier analysis, however, is context-insensitive. Section 3.3 gives a context-sensitive version with refinement [15], referred to here as REFINEPTS. Section 3.4 gives an illustrating example, motivating the need for exploiting local reachability reuse.

3.1 CFL Reachability

Context-free language (CFL) reachability [14, 23] is an extension of graph reachability that is equivalent to the reachability problem formulated in terms of either recursive state machines (RSMs) [3] or set constraints [9]. Let G be a directed graph whose edges are labeled by symbols from an alphabet Σ. Let L be a CFL over Σ. Each path p in G has a string $w(p)$ in Σ^* formed by concatenating in order the labels of edges in p. A node u is *L-reachable* from a node v if there exists a path p from v to u, called an *L-path*, such that $w(p) \in L$.

CFL reachability is computationally more expensive to solve than standard graph reachability. In the case of the single-source L-path problem, which requires finding all nodes L-reachable from a source node n in a graph G, the worst-case time complexity is $O(\Gamma^3 N^3)$, where Γ is the size of a normalized grammar for L and N is the number of nodes in G [14]. Therefore, we are motivated to exploit reachability reuse to lower its analysis overhead in this work.

3.2 Field-Sensitivity

We discuss how to perform field-sensitive points-to analysis without considering context sensitivity in CFL reachability. A context-insensitive analysis merges information from different calls of a method rather than reasoning about each call separately. As a result, global assignment, call entry or call exit edges are all treated as local assignment edges. Given a program, its PAG is thus simplified to possess only four types of local edges: new, assign, load and store.

Let us first consider a PAG G with only new and assign. It suffices to develop a regular language, L_{FT} (FT for flows-to), such that if an object o can flow to a variable v during the execution of the program, then v will be L_{FT}-reachable from o in G. Let *flowsTo* be the start symbol of L_{FT}. Then we have the following (regular) grammar for L_{FT}:

$$\textit{flowsTo} \quad \rightarrow \quad \textsf{new}\ (\ \textsf{assign})^* \tag{1}$$

If o *flowsTo* v, then v is L_{FT}-reachable from o. Thus, we know that o belongs to the points-to set of v.

For field accesses, precise handling of heap accesses is formulated with the updated L_{FT} being a CFL of *balanced parentheses* [16]. Two variables x and y may be aliases if an object o may flow to both x and y. Thus, v may point to o if there exists a pair of statements

$p.f = q$ and $v = u.f$, such that the base variables p and u can be aliases. So o flows through the above two statements with a pair of parentheses (i.e., $\mathsf{store}(f)$ and $\mathsf{load}(f)$), first into q and then into v. Therefore, the *flowsTo* production is extended into:

$$\textit{flowsTo} \quad \rightarrow \quad \mathsf{new}\ (\ \mathsf{assign} \mid \mathsf{store}(f)\ \textit{alias}\ \mathsf{load}(f))^* \qquad (2)$$

where x *alias* y means that x and y could be aliases. To allow *alias* paths in an *alias* language, $\overline{\textit{flowsTo}}$ is introduced as the inverse of the *flowsTo* relation. A *flowsTo*-path p can be inverted to obtain its corresponding $\overline{\textit{flowsTo}}$-path $\overline{p}$ using inverse edges, and vice versa. For each edge $x \xleftarrow{\ell} y$ in p, its inverse edge is $y \xleftarrow{\overline{\ell}} x$ in $\overline{p}$. (To avoid cluttering, the inverse edges in a PAG, such as the one given in Figure 2, are not shown explicitly.) Thus, o *flowsTo* x iff x $\overline{\textit{flowsTo}}$ o. This means that $\overline{\textit{flowsTo}}$ actually represents the standard points-to relation. As a result, x *alias* y iff x $\overline{\textit{flowsTo}}$ o *flowsTo* y for some object o. Thus, the *alias* language is defined by:

$$\begin{array}{rcl} \textit{alias} & \rightarrow & \overline{\textit{flowsTo}}\ \textit{flowsTo} \\ \overline{\textit{flowsTo}} & \rightarrow & (\ \overline{\mathsf{assign}} \mid \overline{\mathsf{load}(f)}\ \textit{alias}\ \overline{\mathsf{store}(f)})^*\ \overline{\mathsf{new}} \end{array} \qquad (3)$$

Our final CFL L_{FT} for finding the points-to set of a variable consists of the productions given in (2) and (3) with $\overline{\textit{flowsTo}}$ as its start symbol. For convenience, we often write *pointsTo* to mean $\overline{\textit{flowsTo}}$.

The RSMs [3] for *pointsTo* and *alias* are shown in Figure 3(a); they will be referred later to facilitate the understanding of DYNSUM.

3.3 Context Sensitivity

A call entry or exit edge is treated as an assign edge as before in L_{FT} to represent parameter passing and method return but assign and assignglobal edges are now distinguished.

A context-sensitive analysis requires call entries and exits to be matched, which is solved also as a balanced-parentheses problem [14]. This is done by filtering out *flowsTo*- and $\overline{\textit{flowsTo}}$-paths corresponding to unrealizable paths. The following CFL R_{RP} (RP for realizable paths) is used to describe all realizable paths in a PAG G; its RSM is given in Figure 3(b):

$$\begin{array}{rcl} C & \rightarrow & \mathsf{CallEntry}_i\ C\ \mathsf{CallExit}_i \ \mid\ C\ C\ \mid \epsilon \\ \mathsf{CallEntry}_i & \rightarrow & \mathsf{entry}_i \mid \overline{\mathsf{exit}_i} \\ \mathsf{CallExit}_i & \rightarrow & \mathsf{exit}_i \mid \overline{\mathsf{entry}_i} \end{array}$$

When traversing a *flowsTo*-path in G, entering a method via entry_i from call site i requires exiting from that method back to call site i via either (1) exit_i to continue its traversal along the same *flowsTo*-path or (2) $\overline{\mathsf{entry}_i}$ to start a new search for a $\overline{\textit{flowsTo}}$-path. The situation for entering a method via $\overline{\mathsf{exit}_i}$ when traversing a $\overline{\textit{flowsTo}}$-path is reversed.

REFINEPTS's context-sensitive analysis [15], given in Algorithms 1 and 2, is to compute CFL reachability for the CFL $L_{\mathrm{REFINEPTS}} = L_{\mathrm{FT}} \cap R_{\mathrm{RP}}$. This is done by tracking the state of R_{RP} for each explored path while computing $\overline{L_{\mathrm{FT}}}$ reachability. As we focus on computing *pointsTo*, i.e., $\overline{\textit{flowsTo}}$ in this paper, a state represents a calling context, which is typically a finite stack configuration corresponding to $\mathsf{CallEntry}_i$ edges.

Given a variable v and a call stack c, SBPOINTSTO(v, c) computes *pointsTo*(v, c), i.e., the points-to set of v in context c. It traverses edges in the reverse direction. Note that for each *flowsTo* edge $x \xleftarrow{\ell} y$, its inverse $\overline{\textit{flowsTo}}$ edge is $y \xleftarrow{\overline{\ell}} x$. Therefore, traversing from x to y along $x \xleftarrow{\ell} y$ in reverse direction means traversing from x to y along $y \xleftarrow{\overline{\ell}} x$. The check for $c = \varnothing$, i.e, ϵ in line 11 allows for partially balanced parentheses (a prefix with unbalanced closed parentheses and a suffix with unbalanced open parentheses) since a realizable path may not start and end in the same method.

Algorithm 1 REFINEPTS's points-to analysis, SBPOINTSTO, for computing $\overline{\textit{flowsTo}}$ [15]. SBFLOWSTO called in line 21, which computes *flowsTo*, is analogous to its "inverse" SBPOINTSTO and thus omitted.

```
SBPointsTo (v, c)
 1: $pts \leftarrow \varnothing$
 2: for each edge $v \xleftarrow{\mathsf{new}} o$ do
 3:     $pts \leftarrow pts \cup \{(o, c)\}$
 4: for each edge $v \xleftarrow{\mathsf{assign}} x$ do
 5:     $pts \leftarrow pts \cup$ SBPointsTo $(x, c)$
 6: for each edge $v \xleftarrow{\mathsf{assignglobal}} x$ do
 7:     $pts \leftarrow pts \cup$ SBPointsTo $(x, \varnothing)$
 8: for each edge $v \xleftarrow{\mathsf{exit}_i} x$ do
 9:     $pts \leftarrow pts \cup$ SBPointsTo $(x, c$.Push$(i))$
10: for each edge $v \xleftarrow{\mathsf{entry}_i} x$ do
11:     if $c$.Peek() $= i$ or $c = \varnothing$ then
12:         $pts \leftarrow pts \cup$ SBPointsTo $(x, c$.Pop$())$
13: for each edge $e = v \xleftarrow{\mathsf{load}(f)} u$ do
14:     for each edge $q \xleftarrow{\mathsf{store}(f)} p$ do
15:         if $e \notin fldsToRefine$ then
16:             $fldsSeen \leftarrow fldsSeen \cup \{e\}$
17:             $pts \leftarrow pts \cup$ SBPointsTo $(p, \varnothing)$
18:         else
19:             $CSalias \leftarrow \varnothing$
20:             for $(o, c') \in$ SBPointsTo $(u, c)$ do
21:                 $CSalias \leftarrow CSalias \cup$ SBFlowsTo$(o, c')$
22:             for $(r, c'') \in CSalias$ do
23:                 if $r = q$ then
24:                     $pts \leftarrow pts \cup$ SBPointsTo $(p, c'')$
25: return $pts$
```

Algorithm 2 The REFINEPTS analysis

```
RefinePTS (v)
26: while true do
27:     $fldsSeen \leftarrow \varnothing$
28:     $pts \leftarrow$ SBPointsTo $(v, \varnothing)$
29:     if satisfyClient$(pts)$ then
30:         return true
31:     else
32:         if $fldsSeen = \varnothing$ then
33:             return false
34:         else
35:             $fldsToRefine \leftarrow fldsToRefine \cup fldsSeen$
```

SBPOINTSTO is context-sensitive for method invocation by matching call entries and exits and also for heap abstraction by distinguishing allocation sites with calling contexts.

Global variables are context-insensitive. As a result, the R_{RP} state is cleared across assignglobal edges (lines 6 and 7). Thus, these edges "skip" the sequence of calls and returns between the reads and writes of a global variable.

To support iterative refinement, REFINEPTS operates with a refinement loop, which is simplified in Algorithm 2 to avoid the complications in dealing with points-to cycles. For more detail, see [15, 16]. Given a points-to query, an initial approximation with a field-based analysis is adopted and then gradually refined until the client is satisfied. In lines 13 and 14, the base variables u and q are assumed to be aliases, if $e = v \xleftarrow{\mathsf{load}(f)} u$ is not in *fldsToRefine*, a set controlling the refinement. In this case, an artificial match edge $v \xleftarrow{\mathsf{match}} p$ is considered to have been introduced. By moving directly from v to p, a sequence of calls and returns between the read and write of field f can be skipped. Hence, the state of R_{RP} is cleared (line 17). If satisfyClient(pts) returns false, then another refinement iteration is needed. All encountered match edges are removed, and the analysis becomes field-sensitive for each such match edge, $v \xleftarrow{\mathsf{match}} p$, so that the paths between their endpoints are explored. This may lead to new match edges to be discovered and further refined until either a pre-set budget is exceeded or the query has been answered (lines 29 and 30).

3.4 A Motivating Example

We explain how REFINEPTS works by using it to compute the points-to sets for $s1$ and $s2$ in Figure 2. We motivate the need for local reachability reuse in DYNSUM in Section 4.

Consider REFINEPTS($s1$) first. To fully resolve its points-to set, the following four iterations are performed:

1. Initially, REFINEPTS starts being field-based since *fldsSeen* = *fldsToRefine* = $\varnothing$. In this first iteration, due to the existence of the match edge, $p \xrightarrow{\mathsf{match}} ret_{\mathrm{get}}$, we find that SBPOINTSTO$(s1, \varnothing) = \{o_{26}, o_{29}\}$ since there are two *flowsTo*-paths: (1) $o_{26} \xrightarrow{\mathsf{new}} tmp1 \xrightarrow{\mathsf{entry}_{26}} p \xrightarrow{\mathsf{match}} ret_{\mathrm{get}} \xrightarrow{\mathsf{exit}_{22}} ret_{\mathrm{retrieve}} \xrightarrow{\mathsf{exit}_{32}} s1$ and (2) $o_{29} \xrightarrow{\mathsf{new}} tmp1 \xrightarrow{\mathsf{entry}_{29}} p \xrightarrow{\mathsf{match}} ret_{\mathrm{get}} \xrightarrow{\mathsf{exit}_{22}} ret_{\mathrm{retrieve}} \xrightarrow{\mathsf{exit}_{32}} s1$.
2. In the second iteration, REFINEPTS starts with *fldsToRefine* $= \{t_{\mathrm{get}} \xrightarrow{\mathsf{load}(arr)} ret_{\mathrm{get}}\}$. There are two new match edges found: $t_{\mathrm{Vector}} \xrightarrow{\mathsf{match}} t_{\mathrm{get}}$ and $t_{\mathrm{Vector}} \xrightarrow{\mathsf{match}} t_{\mathrm{add}}$. As $t_{\mathrm{add}} \xrightarrow{\mathsf{match}} t_{\mathrm{Vector}} \xleftarrow{\mathsf{new}} o_5 \xrightarrow{\mathsf{new}} t_{\mathrm{Vector}} \xrightarrow{\mathsf{match}} t_{\mathrm{get}}$, t_{add} and t_{get} are found to be aliases. Thus, SBPOINTSTO$(s1, \varnothing) = \{o_{26}, o_{29}\}$ remains unchanged.
3. In the third iteration, REFINEPTS continues to refine the two new match edges discovered in the second iteration. SBPOINTSTO starts its traversal from $s1$ along the right part of the graph. Initially, $R_{\mathrm{RP}} = [\![\;]\!]$. On encountering exit_{32} and exit_{22}, the analysis pushes their call sites into the context stack at node ret_{get}: $R_{\mathrm{RP}} = [\![32, 22]\!]$. Then it arrives at t_{retrieve} after having popped the stack once so that $R_{\mathrm{RP}} = [\![32]\!]$. Traversing along another two new match edges, $t_{\mathrm{retrieve}} \xleftarrow{\mathsf{match}} v_{\mathrm{Client}}$ and $t_{\mathrm{retrieve}} \xleftarrow{\mathsf{match}} v_{\mathrm{set}}$, REFINEPTS will next explore from v_{Client} and v_{set}, one by one. As both o_{25} and o_{28} can flow to $this_{\mathrm{Vector}}$ and $this_{\mathrm{add}}$, so $this_{\mathrm{Vector}}$ and $this_{\mathrm{add}}$ are aliases. So once again SBPOINTSTO$(s1, \varnothing) = \{o_{26}, o_{29}\}$ is the same as before.
4. In the last iteration, REFINEPTS continues to refine the two new match edges discovered in the third iteration. Due to context sensitivity, only the edge $this_{\mathrm{retrieve}} \xleftarrow{\mathsf{entry}_{32}} c1$ is realizable because entry_{32} matches the top of context stack $[\![32]\!]$ but $this_{\mathrm{retrieve}} \xleftarrow{\mathsf{entry}_{33}} c2$ does not. Therefore, $this_{\mathrm{Client}}$ and $this_{\mathrm{retrieve}}$ may be aliases. So SBPOINTSTO will eventually visit o_{26} and obtain the final solution: SBPOINTSTO $(s1, \varnothing) = \{o_{26}\}$.

Similarly, s_2 is resolved. However, REFINEPTS will traverse redundantly a few paths that it did before in resolving s_1 in order to conclude that SBPOINTSTO $(s2, \varnothing) = \{o_{29}\}$.

4. THE DYNSUM ANALYSIS

While REFINEPTS may bring benefits for some clients, our motivating example exposes several of its limitations:

- The same paths can be traversed multiple times for a set of queries under the same or different calling contexts. This problem becomes more severe as modern software relies heavily on common libraries (e.g., Java JDK).
- Ad hoc caching techniques [15, 20, 25] are ineffective for three reasons. First, SBPOINTSTO(v, c) cannot be cached unless it is fully resolved within a pre-set budget. Second, the cached SBPOINTSTO(v, c) can only be reused in the same context c. When resolving SBPOINTSTO$(s1, \varnothing)$ and SBPOINTSTO$(s2, \varnothing)$ previously, the points-to set of ret_{get} is computed twice, once for $[\![32, 22]\!]$ and once for $[\![33, 22]\!]$. As a result, the same path from ret_{get} to $this_{\mathrm{get}}$ is still redundantly traversed for such different contexts. Finally, caching and refinement may be incompatible as a cached points-to set may depend on the match edges encountered when the points-to set was computed.
- All field-based refinement iterations are pure overhead before a client can be satisfied with a particular query. This "lazy" strategy is not well-suited for clients that require precise points-to or aliasing information.

In this work, we propose to overcome these limitations by giving up refinement and relying on exploiting local reachability reuse to efficiently answer demand queries. As shown in Figure 2, we distinguish two types of edges in a PAG: *local edges* (new, assign, load and store) and *global edges* (assignglobal, entry_i and exit_i). The key observation is that local edges have no effects on the context of a query while global edges have no effects on its field-sensitivity.

Therefore, our DYNSUM analysis is broken down into two parts. DSPOINTSTO given in Algorithm 3 performs a *partial points-to analysis* (PPTA) on-the-fly for a queried variable to summarize its points-to relations along the local edges within a method field-sensitively but context-independently. DYNSUM in Algorithm 4 handles the context-dependent global edges while collaborating with PPTA to compute new summaries if they are unavailable for reuse.

4.1 PPTA: Partial Points-to Analysis

It is easy to understand what PPTA is in terms of the RSMs given in Figure 3, as the two RSMs (for *pointsTo* and *alias*) in Figure 3(a), which are together equivalent to L_{FT}, handle field-sensitivity, and the RSM for R_{RP} shown in Figure 3(b) handles context-sensitivity.

PPTA aims to summarize all state transitions field-sensitively but context-insensitively made along the local edges of a method according to the *pointsTo* and *alias* RSMs given in Figure 3(a). Starting with a points-to query for a variable v in context c, we will

Algorithm 3 PPTA-based summarization

```
DSPointsTo ($v, f, s, visited$)
1:  if $(v, f, s) \in visited$ then
2:      return $\varnothing$
3:  $visited \leftarrow visited \cup \{ (v, f, s) \}$
4:  $pts \leftarrow \varnothing$
5:  if $s = S_1$ then
6:      for each edge $v \xleftarrow{\text{new}} o$ do
7:          if $f = \varnothing$ then
8:              $pts \leftarrow pts \cup \{ o \}$
9:          else
10:             $pts \leftarrow pts \cup$ DSPointsTo $(v, f, S_2, visited)$
11:     for each edge $v \xleftarrow{\text{assign}} x$ do
12:         $pts \leftarrow pts \cup$ DSPointsTo $(x, f, S_1, visited)$
13:     for each edge $v \xleftarrow{\text{load}(g)} x$ do
14:         $pts \leftarrow pts \cup$ DSPointsTo $(x, f.\text{Push}(\overline{g}), S_1, visited)$
15:     if $v$ has a global edge flowing into $v$ then
16:         $pts \leftarrow pts \cup \{ (v, f, S_1) \}$
17: if $s = S_2$ then
18:     for each edge $x \xleftarrow{\text{load}(g)} v$ do
19:         if $f$.Peek() $= g$ then
20:             $pts \leftarrow pts \cup$ DSPointsTo $(x, f.\text{Pop}(), S_2, visited)$
21:     for each edge $x \xleftarrow{\text{assign}} v$ do
22:         $pts \leftarrow pts \cup$ DSPointsTo $(x, f, S_2, visited)$
23:     for each edge $x \xleftarrow{\text{store}(g)} v$ do
24:         $pts \leftarrow pts \cup$ DSPointsTo $(x, f.\text{Push}(g), S_1, visited)$
25:     for each edge $v \xleftarrow{\text{store}(g)} x$ do
26:         if $f$.Peek() $= \overline{g}$ then
27:             $pts \leftarrow pts \cup$ DSPointsTo $(x, f.\text{Pop}(), S_1, visited)$
28:     if $v$ has a global edge flowing out of $v$ then
29:         $pts \leftarrow pts \cup \{ (v, f, S_2) \}$
30: return $pts$
```

eventually arrive at the two RSMs with a new query (u, f, s), where u is a node in some method m, f is a *field stack* containing the field edge labels encountered but not yet matched, and s is a state indicating the *direction* in which the analysis traverses—along a $\overline{flowsTo}$ path if $s = S_1$ and a *flowsTo* path if $s = S_2$. The objective of performing PPTA for (u, f, s) is to compute a so-called *partial points-to set* for u, denoted ppta(u, f, s), so that (1) ppta(u, f, s) contains all objects o in method m that flow to u, and (2) all tuples (u', f', s') eventually reached by the *pointsTo* and *alias* RSMs given in Figure 3(a) along only the local edges in method m. Each such tuple represents a state reached this way and will be cached for later reuse just before a global edge is about to be traversed.

Consider our example given in Figure 2 again. We have ppta$(ret_{\text{get}}, \varnothing, S_1) = \{(this_{\text{get}}, [\![\overline{arr}, \overline{elems}]\!], S_1)\}$, which shows intuitively that the points-to set of $this_{\text{get}}.elems.arr$ must be included in the points-to set of ret_{get}. Note that this PPTA information is computed when answering the points-to query for $s1$ and will be reused later when the points-to query $s2$ is answered.

For another example, suppose we want to compute the points-to set for $s2$ with an empty context. By traversing the right part of the PAG in Figure 2, we will eventually need to compute a query for $(this_{\text{set}}, [\![\overline{arr}, \overline{elems}, \overline{vec}]\!], S_2)$ (as later illustrated in Steps 6 – 7 for $s2$ in Table 1). By performing a PPTA, we find that ppta$(this_{\text{set}}, [\![\overline{arr}, \overline{elems}, \overline{vec}]\!], S_2) = \{(v_{\text{set}}, [\![\overline{arr}, \overline{elems}]\!], S_1)\}$.

Algorithm 4 The DynSum analysis

```
DynSum ($v, c$)
1:  $pts \leftarrow \varnothing$
2:  $w \leftarrow \{ (v, \varnothing, S_1, c) \}$
3:  while $w \neq \varnothing$ do
4:      remove $(u, f, s, c')$ from $w$
5:      if $((u, f, s), l) \in Cache$ then
6:          $ppta \leftarrow l$
7:      else
8:          $ppta \leftarrow$ DSPointsTo $(u, f, s, \varnothing)$
9:          $Cache \leftarrow Cache \cup ((u, f, s), ppta)$
10:     for each $o \in ppta$ do
11:         $pts \leftarrow pts \cup \{ (o, c') \}$
12:     for each $(x, f', s') \in ppta$ do
13:         if $s' = S_1$ then
14:             for each $x \xleftarrow{\text{exit}_i} y$ do
15:                 Propagate$(w, y, f', S_1, c'$.Push$(i))$
16:             for each edge $x \xleftarrow{\text{entry}_i} y$ do
17:                 if $c' = \varnothing$ or $c'$.Peek()$= i$ then
18:                     Propagate$(w, y, f', S_1, c'$.Pop())
19:             for each edge $x \xleftarrow{\text{assignglobal}} y$ do
20:                 Propagate$(w, y, f', S_1, \varnothing)$
21:         if $s' = S_2$ then
22:             for each edge $y \xleftarrow{\text{exit}_i} x$ do
23:                 if $c' = \varnothing$ or $c'$.Peek()$= i$ then
24:                     Propagate$(w, y, f', S_2, c'$.Pop())
25:             for each edge $y \xleftarrow{\text{entry}_i} x$ do
26:                 Propagate$(w, y, f', S_2, c'$.Push$(i))$
27:             for each edge $y \xleftarrow{\text{assignglobal}} x$ do
28:                 Propagate$(w, y, f', S_2, \varnothing)$
29: return $pts$

Propagate($w, n, f, s, c$)
1:  if $(n, f, s, c) \notin w$ then
2:      $w \leftarrow w \cup \{ (n, f, s, c) \}$
```

4.2 Algorithms

Algorithm 3. This is a recursive algorithm that propagates the context-independent CFL-reachability information across a given PAG. There can be points-to cycles in a PAG. Therefore, the set $visited$ of visited nodes is used to avoid re-traversing a cycle more than once, as in [15].

The analysis strictly follows the *pointsTo* and *alias* RSMs for L_{FT} given in Figure 3(a), which has two states, S_1 and S_2. All transitions on S_1 are handled in lines 5 – 16 and those on S_2 in lines 17 – 29. Let us consider S_1 first. On encountering an edge $v \xleftarrow{\text{new}} o$ (lines 6 – 10), the analysis will insert the object o into pts only when the field stack f is empty. Otherwise, it will traverse a $\overline{flowsTo}$ path to find an *alias* relation between v and some x such that v *alias* x holds. An alias relation is discovered by following the *alias* RSM given in Figure 3(a). In lines 11 – 14, the assign and load edges are handled. In lines 15 – 16, on encountering a global edge, PPTA stores the current state in pts. Lines 17 – 29 for dealing with state S_2 are similar. The only interesting part happens in lines 25 – 27, which accepts a store edge when the top of the field stack f matches the label of the store edge, $\overline{g}$.

Note that the two states S_1 and S_2 are handled asymmetrically since the *alias* RSM in Figure 3(a) is "asymmetric", or precisely, is recursive. There are four cases involved in handling field accesses: $\mathsf{load}(g)$, $\mathsf{store}(g)$, $\overline{\mathsf{load}(g)}$ and $\overline{\mathsf{store}(g)}$. In the PPTA algorithm, the $\overline{\mathsf{load}(g)}$ edges are handled in S_1 while the other three in S_2. In S_1, the *alias* RSM will process a $\overline{\mathsf{load}(g)}$ edge, $v \xleftarrow{\overline{\mathsf{load}(g)}} x$, and stay in S_1. In S_2, the *alias* RSM will process (1) a $\mathsf{load}(g)$ edge, $x \xleftarrow{\mathsf{load}(g)} v$, and stay in S_2, (2) a $\mathsf{store}(g)$ edge, $x \xleftarrow{\mathsf{store}(g)} v$, and then transit to S_1 to look for aliases for the base variable x of the store, and (3) a $\overline{\mathsf{store}(g)}$ edge, $v \xleftarrow{\overline{\mathsf{store}(g)}} x$, and transit to S_1 if the base variable v is an alias of the base variable of the most recent load processed earlier in lines 13 – 14. Note that the *alias* RSM can only move from S_1 to S_2 at an allocation site on $\overline{\mathsf{new}}$ new, i.e., by first traversing the corresponding $\overline{\mathsf{new}}$ edge and then the same edge in the opposite direction, which is the new edge.

Algorithm 4. This is where our DynSum analysis starts. When called, DynSum (v, c) will return the points-to set of a queried variable v in context c. This is a worklist algorithm that propagates the CFL-reachability facts through a given PAG. Because the local edges are handled as a PPTA by Algorithm 3, Algorithm 4 deals with only the context-dependent global edges according to the RSM R_{RP} in Figure 3(b) while calling Algorithm 3 to perform all required PPTA steps.

Each worklist element is a tuple of the form (u, f, s, c), indicating that the computation for v has reached node u, where u is a new queried variable generated, with the current field stack f, the current "direction" state $s \in \{S1, S2\}$ of the RSM given in Figure 3(a) and the current context stack c. In lines 5 – 9, the summary $ppta$ for the query (u, f, s) is reused if it is available in $Cache$ and computed otherwise by calling Algorithm 3. As $ppta$ returned from PPTA contains both objects and tuples, DynSum handles objects in lines 10 – 11 and tuples in lines 12 – 28. The $\mathsf{assignglobal}$, exit_i and entry_i edges are handled according to the RSM for R_{RP} given in Figure 3(b), similarly as in RefinePTS.

4.3 Example

We highlight the advantages of DynSum using the example given in Figure 2. In our implementation of Algorithm 4, DSPointsTo is not called in line 8 to perform the PPTA if u has no local edges.

Suppose we want to answer the same two points-to queries $s1$ and $s2$ as before. Table 1 illustrates how local reachability reuse is exploited in our analysis by showing only the traversed edges that lead directly to their points-to targets: o_{26} for $s1$ and o_{29} for $s2$.

Suppose $s1$ is issued first and then followed by $s2$. DynSum starts from $s1$ with the initial state being $(s1, \varnothing, S_1, \varnothing)$. The analysis encounters the incoming $\overline{\mathsf{exit}_{32}}$ edge, staying at S_1 and pushing 32 into the context stack. The new state is $(ret_{\mathrm{retrieve}}, \varnothing, S_1, [\![32]\!])$.

Next, DynSum processes edges according to the RSMs given in Figures 3(a) and (b). On encountering a node with some local edges, the analysis first performs a PPTA on the node and then uses its summarized partial points-to set to continue its exploration. If the summarized partial points-to set is available in the cache, then it is reused straightaway to speed up the exploration.

Finally, DynSum reaches $tmp1 \xleftarrow{\mathsf{new}} o_{26}$, by completing its analysis in 23 steps. The points-to set of $s1$ is $\{o_{26}\}$.

Step	v	f	s	c	Edge
0	$s1$	$[\![\ ]\!]$	S_1	$[\![\ ]\!]$	$\overline{\mathsf{exit}_{32}}$
1	ret_{retrieve}	$[\![\ ]\!]$	S_1	$[\![32]\!]$	$\overline{\mathsf{exit}_{22}}$
2	ret_{get}	$[\![\ ]\!]$	S_1	$[\![32, 22]\!]$	$\overline{\mathsf{load}(a)}$
3	t_{get}	$[\![\overline{a}]\!]$	S_1	$[\![32, 22]\!]$	$\overline{\mathsf{load}(e)}$
4	$this_{\mathrm{get}}$	$[\![\overline{a}, \overline{e}]\!]$	S_1	$[\![32, 22]\!]$	$\overline{\mathsf{entry}_{22}}$
5	t_{retrieve}	$[\![\overline{a}, \overline{e}]\!]$	S_1	$[\![32]\!]$	$\overline{\mathsf{load}(v)}$
6	$this_{\mathrm{retrieve}}$	$[\![\overline{a}, \overline{e}, \overline{v}]\!]$	S_1	$[\![32]\!]$	$\overline{\mathsf{entry}_{32}}$
7	$c1$	$[\![\overline{a}, \overline{e}, \overline{v}]\!]$	S_1	$[\![\ ]\!]$	$\overline{\mathsf{new}}\ \mathsf{new}$
8	$c1$	$[\![\overline{a}, \overline{e}, \overline{v}]\!]$	S_2	$[\![\ ]\!]$	entry_{27}
9	$this_{\mathrm{Client}}$	$[\![\overline{a}, \overline{e}, \overline{v}]\!]$	S_2	$[\![27]\!]$	$\overline{\mathsf{store}(v)}$
10	v_{Client}	$[\![\overline{a}, \overline{e}]\!]$	S_1	$[\![27]\!]$	$\overline{\mathsf{entry}_{27}}$
11	$v1$	$[\![\overline{a}, \overline{e}]\!]$	S_1	$[\![\ ]\!]$	$\overline{\mathsf{new}}\ \mathsf{new}$
12	$v1$	$[\![\overline{a}, \overline{e}]\!]$	S_2	$[\![\ ]\!]$	entry_{25}
13	$this_{\mathrm{Vector}}$	$[\![\overline{a}, \overline{e}]\!]$	S_2	$[\![25]\!]$	$\overline{\mathsf{store}(e)}$
14	t_{Vector}	$[\![\overline{a}]\!]$	S_1	$[\![25]\!]$	$\overline{\mathsf{new}}\ \mathsf{new}$
15	t_{Vector}	$[\![\overline{a}]\!]$	S_2	$[\![25]\!]$	$\mathsf{store}(e)$
16	$this_{\mathrm{Vector}}$	$[\![\overline{a}, e]\!]$	S_1	$[\![25]\!]$	$\overline{\mathsf{entry}_{25}}$
17	$v1$	$[\![\overline{a}, e]\!]$	S_1	$[\![\ ]\!]$	$\overline{\mathsf{new}}\ \mathsf{new}$
18	$v1$	$[\![\overline{a}, e]\!]$	S_2	$[\![\ ]\!]$	entry_{26}
19	$this_{\mathrm{add}}$	$[\![\overline{a}, e]\!]$	S_2	$[\![26]\!]$	$\mathsf{load}(e)$
20	t_{add}	$[\![\overline{a}]\!]$	S_2	$[\![26]\!]$	$\overline{\mathsf{store}(a)}$
21	p	$[\![\ ]\!]$	S_1	$[\![26]\!]$	$\overline{\mathsf{entry}_{26}}$
22	$tmp1$	$[\![\ ]\!]$	S_1	$[\![\ ]\!]$	$\overline{\mathsf{new}}$
23	o_{26}	$[\![\ ]\!]$	S_1	$[\![\ ]\!]$	
0	$s2$	$[\![\ ]\!]$	S_1	$[\![\ ]\!]$	$\overline{\mathsf{exit}_{33}}$
1	ret_{retrieve}	$[\![\ ]\!]$	S_1	$[\![33]\!]$	$\overline{\mathsf{exit}_{22}}$
2	ret_{get}	$[\![\ ]\!]$	S_1	$[\![33, 22]\!]$	
	$\overline{\mathsf{load}(a)}\ \overline{\mathsf{load}(e)}$				**reuse**
	$this_{\mathrm{get}}$	$[\![\overline{a}, \overline{e}]\!]$	S_1	$[\![33, 22]\!]$	$\overline{\mathsf{entry}_{22}}$
3	t_{retrieve}	$[\![\overline{a}, \overline{e}]\!]$	S_1	$[\![33]\!]$	
	$\overline{\mathsf{load}(v)}$				**reuse**
	$this_{\mathrm{retrieve}}$	$[\![\overline{a}, \overline{e}, \overline{v}]\!]$	S_1	$[\![33]\!]$	$\overline{\mathsf{entry}_{33}}$
4	$c2$	$[\![\overline{a}, \overline{e}, \overline{v}]\!]$	S_1	$[\![\ ]\!]$	$\overline{\mathsf{new}}\ \mathsf{new}$
5	$c2$	$[\![\overline{a}, \overline{e}, \overline{v}]\!]$	S_2	$[\![\ ]\!]$	entry_{31}
6	$this_{\mathrm{set}}$	$[\![\overline{a}, \overline{e}, \overline{v}]\!]$	S_2	$[\![31]\!]$	$\overline{\mathsf{store}(v)}$
7	v_{set}	$[\![\overline{a}, \overline{e}]\!]$	S_1	$[\![31]\!]$	$\overline{\mathsf{entry}_{31}}$
8	$v2$	$[\![\overline{a}, \overline{e}]\!]$	S_1	$[\![\ ]\!]$	$\overline{\mathsf{new}}\ \mathsf{new}$
9	$v2$	$[\![\overline{a}, \overline{e}]\!]$	S_2	$[\![\ ]\!]$	entry_{28}
10	$this_{\mathrm{Vector}}$	$[\![\overline{a}, \overline{e}]\!]$	S_2	$[\![28]\!]$	
	$\overline{\mathsf{store}(e)}\ \overline{\mathsf{new}}\ \mathsf{new}\ \mathsf{store}(e)$				**reuse**
	$this_{\mathrm{Vector}}$	$[\![\overline{a}, e]\!]$	S_1	$[\![28]\!]$	$\overline{\mathsf{entry}_{28}}$
11	$v2$	$[\![\overline{a}, e]\!]$	S_1	$[\![\ ]\!]$	$\overline{\mathsf{new}}\ \mathsf{new}$
12	$v2$	$[\![\overline{a}, e]\!]$	S_2	$[\![\ ]\!]$	entry_{29}
13	$this_{\mathrm{add}}$	$[\![\overline{a}, e]\!]$	S_2	$[\![29]\!]$	
	$\mathsf{load}(e)\ \overline{\mathsf{store}(a)}$				**reuse**
	p	$[\![\ ]\!]$	S_1	$[\![29]\!]$	$\overline{\mathsf{entry}_{29}}$
14	$tmp2$	$[\![\ ]\!]$	S_1	$[\![\ ]\!]$	$\overline{\mathsf{new}}$
15	o_{29}	$[\![\ ]\!]$	S_1	$[\![\ ]\!]$	

Table 1: Traversals of DynSum when answering the points-to queries for $s1$ and $s2$ in our motivating example (a, e and v stand for fields arr, $elems$ and $vector$, respectively).

When $s2$ is issued, the summaries computed earlier can be reused. As shown in the bottom part of Table 1, DynSum takes only 15 steps to find $\{o_{29}\}$ as its points-to set. Ad hoc caching techniques [15, 20, 25] are not helpful since both queries require different calling contexts to be traversed, as explained earlier.

Algorithm	Full Precision	Memorization	Reuse	On-Demandness
NoRefine	Yes	No	No	Yes
RefinePTS	Yes	Dynamic (within queries)	Context Dependent	Yes
StaSum	No	Static (across queries)	Context Independent	Partly
DynSum	Yes	Dynamic (across queries)	Context Independent	Yes

Table 2: Strengths and weaknesses of four demand-driven points-to analyses.

For this example, the summaries computed during query $s1$ are not reused within in the same query. In general, however, reuse can happen both within a query and during subsequent queries.

4.4 Comparison

We compare four context- and field-sensitive demand-driven points-to or alias analyses in Table 2 now and in our evaluation later:

- RefinePTS. This is the algorithm from [15] with an open-source release. As reviewed earlier, RefinePTS uses a refinement policy to satisfy a client's queries. All queries are handled independently. Ad hoc caching is used to avoid unnecessary traversals within a query.
- NoRefine. This is the version of RefinePTS with neither refinement nor ad hoc caching.
- StaSum. This is the algorithm introduced in [22], which computes all-pair reachability summaries for each method off-line and then reuses the summaries to accelerate demand queries. In our experiments, such summaries are computed for all methods on the PAG instead of a symbolic graph of the program. No efforts are made to avoid some summaries based on some user-supplied heuristics.
- DynSum. This is the one introduced in this paper. DynSum can deliver the same precision as RefinePTS with enough budgets and is fully on-demand without performing any unnecessary computations to achieve great reuse.

5. EVALUATION

We evaluate the efficiency of DynSum by comparing it with RefinePTS using nine Java benchmarks, selected from the Dacapo and SPECjvm98 benchmark suites. For reference purposes, the performance of NoRefine is also given. As StaSum is not available to us, we will compare it with DynSum in terms of the number of summaries computed. Our evaluation has validated the following two experimental hypotheses about the proposed DynSum approach:

- **DynSum is more scalable than RefinePTS.** DynSum outperforms RefinePTS by 1.95×, 2.28× and 1.37× on average for the three clients discussed below.

  DynSum avoids a great number of unnecessary computations and thus represents a good optimization for context-sensitive demand-driven analysis.
- **DynSum is more scalable than StaSum.** DynSum computes significantly fewer summaries than StaSum for the same three clients, making it better-suited for low-budget environments like JIT compilers and IDEs.

5.1 Implementation

RefinePTS is publicly available in the Soot 2.4.0 [18] and Spark [10] frameworks. We have implemented DynSum and NoRefine in the same frameworks and conducted our experiments using the Sun JDK 1.6.0_16 libraries. Unmodeled native methods and reflection calls [12, 21] are handled conservatively and Tamiflex [2] is used. As all three analyses are context-sensitive, the call graph of the program is constructed on-the-fly so that a *context-sensitive* call graph is always maintained during the CFL-reachability exploration.

When introducing all three algorithms earlier, we have assumed cycle-free PAGs to make them easy to understand. However, recursion is handled as described in [15] by computing the call graph on-the-fly with recursion cycles collapsed. Points-to cycles are also handled using visited flags in Algorithm 3 as described in [15] by ensuring that a node is not cyclically visited.

5.2 Methodology

We have conducted our experiments on a machine consisting of four AMD Opteron 2.2GHz processors (12 cores each) with 32 GB memory, running RedHat Enterprise Linux 5 (kernel version 2.6.18). Although the system has multi-cores, each analysis algorithm is single-threaded.

We have selected the following three representative clients:

- `SafeCast`. This client checks the safety of downcasts in a program as also discussed in [15].
- `NullDeref`. This client detects null pointer violations, demanding high precision from points-to analysis.
- `FactoryM`. This client checks that a factory method returns a newly-allocated object for each call as in [15].

The benchmarks we used for evaluation are nine Java programs selected from the SPECjvm98 and Dacapo benchmark suites. Table 3 shows the number of different kinds of nodes and edges in the context-sensitive PAG of a program. The *locality* of a PAG is measured as the percentage of local (*flowsTo*) edges (including new, assign, load and store) among all (*flowsTo*) edges. This metric is used to demonstrate the scope of our optimization. As can be seen from Table 3, the majority of the edges in a PAG are local edges. This implies that a large number of paths with only local edges can be summarized in context-independent manner and reused later.

In the last three columns, the total number of queries issued by a client in a program is given. Each client continuously issues points-to queries to an analysis. A query is either positively answered by the analysis or terminated once a pre-set budget is exceeded. In our experiments, we have also carefully divided the queries from a client into batches to demonstrate the scalability of DynSum compared to RefinePTS and StaSum as the number of queries increases.

Benchmark	#Methods	#Nodes (K)		#Edges (K)							Locality	#Queries		
	(K)	O	$(V \cup G)$	new	assign	load	store	entry	exit	assignglobal		`SafeCast`	`NullDeref`	`FactoryM`
jack	0.5	16.6	207.9	16.6	328.1	25.1	8.8	39.9	12.8	2.4	87.3%	134	356	127
javac	1.1	17.2	216.1	17.2	367.4	26.8	9.1	42.4	13.3	0.5	88.2%	307	2897	231
soot-c	3.4	9.4	104.8	9.4	195.1	13.3	4.2	19.3	6.4	0.7	89.4%	906	2290	619
bloat	2.2	10.3	115.2	10.3	217.2	14.5	4.6	20.6	6.1	1.0	89.9%	1217	3469	613
jython	3.2	9.5	109.0	9.5	168.4	14.4	4.2	19.5	7.1	1.3	87.6%	464	3351	214
avrora	1.6	4.5	45.1	4.5	38.1	6.0	2.9	9.7	2.9	0.3	80.0%	1130	4689	334
batik	2.3	10.8	118.1	10.8	119.7	13.4	5.3	24.8	7.8	0.6	81.8%	2748	5738	769
luindex	1.0	4.4	48.2	4.4	42.6	6.9	2.3	9.1	3.0	0.5	81.7%	1666	4899	657
xalan	2.5	6.6	75.8	6.6	76.4	14.1	4.4	15.7	4.0	0.2	83.6%	4090	10872	1290

Table 3: Benchmark statistics. Note that Column "O (objs)" is identical to Column "new". All of the numbers include the reachable parts of the Java library, determined using a call graph constructed on the fly with Andersen-style analysis [1] by Spark [10]. Column "*locality*" gives the ratio of local edges among all edges in a PAG. The last three columns give the number of queries issued by each client for a program.

We repeated each experiment three times and reported the average time of the three runs, which includes the time elapsed on points-to analysis and client analysis. All the experiments have low variance in performance. For all analysis algorithms compared, the budget limitation is 75,000, indicating the maximum number of edges that can be traversed in a PAG in order to answer one points-to query.

5.3 Results and Analysis

Analysis Times. Table 4 compares the analysis times of DynSum with RefinePTS and NoRefine for the three clients. NoRefine is the slowest in most cases but can be faster than RefinePTS in some benchmarks for clients `SafeCast` and `NullDeref`. In contrast, DynSum is always faster than NoRefine in all benchmarks for all three clients.

Let us compare DynSum and RefinePTS. DynSum is only slightly slower in `avrora` for `SafeCast` and `luindex` for `FactoryM`. DynSum attains its best performance in `soot-c` for `NullDeref`, outperforming RefinePTS by 4.19×. The average speedups achieved by DynSum for the three clients `SafeCast`, `NullDeref` and `FactoryM` are 1.95×, 2.28× and 1.37×, respectively.

The client that benefits the most from DynSum is `NullDeref`, which requires more precision than the other two clients. Given such high-precision requirements, RefinePTS can hardly terminate early, effectively rendering its repeated refinement steps as pure overhead. This fact is also reflected by the similar analysis times taken by both RefinePTS and NoRefine for this client.

As garbage collection is enabled, it is difficult to monitor memory usage precisely. In our all experiments, DynSum never exceeds 20% more than RefinePTS in terms of the peak memory usage.

Scalability in Answering Demand Queries. We have selected `soot-c`, `bloat` and `jython` to demonstrate that DynSum is more scalable than RefinePTS and StaSum. These applications are selected because they have large code bases, i.e., large PAGs and also a great number of queries issued as shown in Table 3. For each program, we divide the sequence of queries issued by a client into 10 batches. If a client has n_q queries, then each of the first nine batches contains $\lfloor n_q/10 \rfloor$ queries and the last one gets the rest.

- **Comparing with RefinePTS** Figure 4 compares the times taken by DynSum for handling each batch of queries normalized with respect to RefinePTS. As more batches are processed, more points-to relations will have been summarized dynamically and recorded for later reuse, and consequently, the less time that DynSum takes to process each subsequent batch.
- **Comparing with StaSum** We collect the number of summaries computed by DynSum at the end of each batch and compare it with StaSum for the three selected benchmarks. For DynSum, the number of summaries computed is available as the size of $Cache$ given in Algorithm 4. For StaSum, all possible summaries for each call entry or exit in a PAG are computed. While StaSum can reduce its number of such summaries based on a user-supplied threshold [22], it is unclear how this can be done effectively by the user, particularly when its optimal value varies from program to program.

  Figure 5 compares the (cumulative) size of summaries computed by DynSum normalized with respect to StaSum. DynSum only needs to compute 41.3%, 47.7% and 37.3% of the summaries computed by StaSum on average in order to handle all the queries issued by the three clients. Furthermore, the number of summaries increases dynamically as the number of queries increases, highlighting the dynamic nature of DynSum.

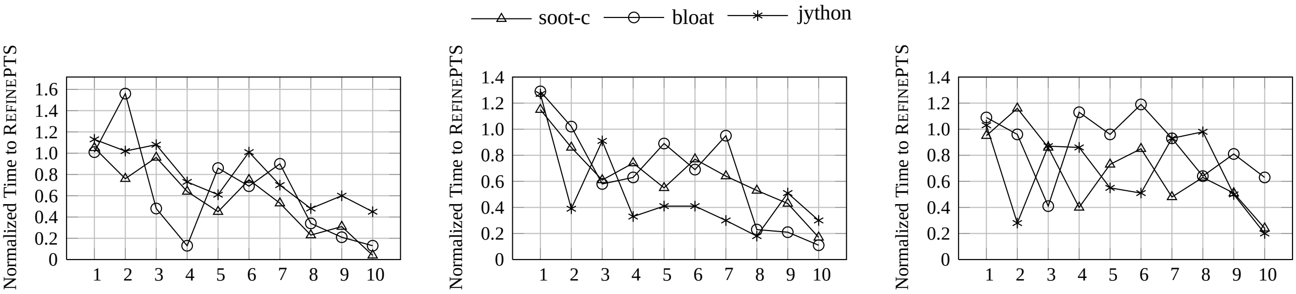
Through these studies, we find that DynSum is effective in avoiding unnecessary traversals made as in RefinePTS and unnecessary summaries computed as in StaSum. The increased scalability makes DynSum better-suited to low-budget environments such as JIT compilers and IDEs in which software may undergo a lot of changes.

6. RELATED WORK

In recent years, there has been a large body of research devoted to points-to analysis, with the summary-based approach to be the most popular and general for achieving context sensitivity. However,

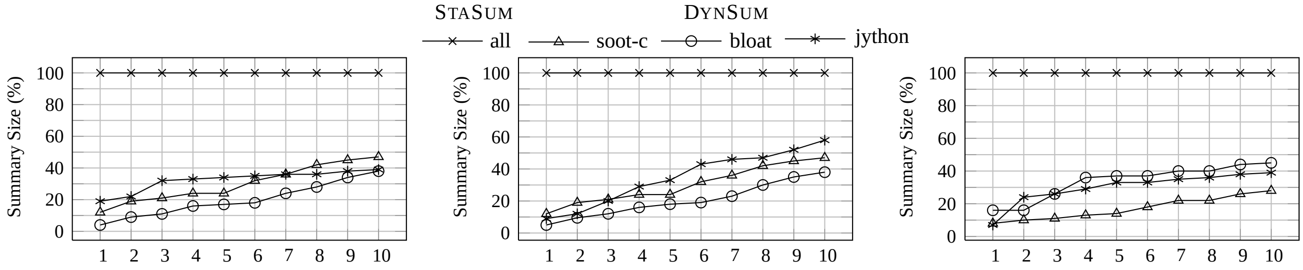
		jack	javac	soot-c	bloat	jython	avrora	batik	luindex	xalan
`SafeCast`	NoRefine	31.0	68.1	134.7	68.2	61.8	39.1	43.4	47.6	459.1
	RefinePTS	28.4	77.9	127.9	76.3	50.9	30.2	29.8	44.9	457.5
	DynSum	15.2	41.3	37.5	32.8	32.2	35.1	19.7	25.3	194.5
`NullDeref`	NoRefine	121.0	174.4	212.3	72.8	160.0	84.4	95.0	57.1	797.9
	RefinePTS	145.6	163.9	221.0	73.5	150.2	20.6	80.7	60.1	575.7
	DynSum	52.6	87.5	52.8	42.6	72.3	13.6	46.4	41.3	194.1
`FactoryM`	NoRefine	26.3	85.1	22.8	147.1	15.7	30.1	41.2	20.7	139.1
	RefinePTS	25.4	60.5	9.5	104.6	15.4	27.9	33.9	13.1	117.8
	DynSum	23.4	47.2	6.7	75.1	6.3	24.4	24.3	13.4	99.5

Table 4: Analysis times of NoRefine, RefinePTS and DynSum for the three clients: `SafeCast`, `NullDeref` and `FactoryM`.



(a) `SafeCast` (b) `NullDeref` (c) `FactoryM`

Figure 4: Normalized analysis times for each batch of queries normalized with respect to RefinePTS.



(a) `SafeCast` (b) `NullDeref` (c) `FactoryM`

Figure 5: The cumulative number of summaries computed by DynSum normalized with respect to StaSum.

existing summary-based algorithms [13, 17, 19, 24] are mostly whole-program-based. How to compute summaries efficiently for demand-driven analysis is less well-understood. Below we focus only on the work directly related to demand-driven points-to analysis.

To accelerate demand queries, some techniques to speed up demand-driven points-to analysis have been explored. In the refinement-based approach introduced in [15], the analysis starts to be field-based for all heap accesses and introduces gradually field-sensitivity into those heap accesses where a better precision may be obtained. In [20], a (whole-program) pre-analysis is presented to improve the performance of demand-driven points-to analysis in Java. In demand-driven analysis techniques [15, 20, 25], budget limitation is commonly used to give a conservative answer for a query once a pre-set budget has been exceeded.

Reps et al. [11, 14] pioneered the research on program analysis via graph reachability. They formulate a number of static analysis programs in terms of CFL reachability, leading to a natural solution to demand-driven points-to analysis.

Heintze and Tardieu [8] introduced a deduction-based demand-driven points-to analysis for C to determine the points-to sets based on demand queries from a client.

Sridharan et al. [15, 16] have proposed two approaches to solving CFL-reachability-based demand-driven points-to analysis for Java. They initially presented a CFL-reachability formulation to model heap accesses as a balanced-parentheses problem in a context-insensitive manner [16]. Later, they extended this earlier work to obtain a context-sensitive points-to analysis [15]. The starting point of our PPTA-based solution, DynSum, is Sirdharan and Bodik's refinement-based analysis [15], using Spark's PAG [10] as our program representation. DynSum improves the performance of this state-of-the-art work significantly without losing precision.

Zheng and Rugina [25] described a demand-driven alias analysis for C. Unlike Heintze and Tardieu's analysis [8], Zheng and Rugina's analysis relies a memory alias CFL reachability formulation. Their analysis is context-insensitive with indirect function calls being conservatively handled. As a result, realizable and unrealizable paths are not distinguished, resulting in both precision and performance loss for some queries.

Xu et al. [20] proposed a pre-analysis to speed up the context-sensitive points-to analysis introduced in [15]. The analysis builds a symbolic graph to reduce the size of a program's PAG but it is whole-program-based.

Yan et al. [22] have recently extended the work of [20] to perform a demand-driven alias analysis without having to compute points-to sets. The proposed approach, denoted StaSum, is compared with DynSum in Table 2 and Figure 5.

Some existing techniques [15, 20, 25] on memorization are ad hoc, limiting their scope and effectiveness. The points-to set $pts(v, c)$ of a variable v in a calling context c is cached only after all v's pointed-to objects have been fully resolved, which does not happen once a pre-set budget has been exceeded. Due to such full reachability reuse, $pts(v, c)$ can only be reused for v in exactly the same (full) context c. In addition, these existing memorization techniques do not directly apply to the state-of-the-art refinement-based approach [15] since the underlying PAG may change due to the iterative refinement used. To the best of our knowledge, this work represents the first systematic investigation on how to exploit local reachability reuse dynamically in order to improve the performance of context-sensitive demand-driven points-to analysis in CFL reachability.

7. CONCLUSION

In this paper, we investigate how to dynamically exploit local reachability reuse to improve the performance of CFL-reachability based demand-driven points-to analysis. Evaluation and validation using three client applications over a range of nine Java benchmarks show that our PPTA-based approach can significantly boost the performance of a state-of-the-art demand-driven points-to analysis without any precision loss. Our approach is particularly useful in low-budget environments such as JIT compilers and IDEs, especially when the program undergoes constantly a lot of changes.

8. ACKNOWLEDGEMENTS

This research is supported by a grant from Oracle Labs and also an Australian Research Council Grant DP0987236.

9. REFERENCES

[1] L. O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[2] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE '11*, pages 241–250, 2011.

[3] S. Chaudhuri. Subcubic algorithms for recursive state machines. In *POPL '08*, pages 159–169, 2008.

[4] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *PLDI '98*, pages 85–96, 1998.

[5] B. Hardekopf and C. Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI '07*, pages 290–299, 2007.

[6] B. Hardekopf and C. Lin. Semi-sparse flow-sensitive pointer analysis. In *POPL '09*, pages 226–238, 2009.

[7] B. Hardekopf and C. Lin. Flow-sensitive pointer analysis for millions of lines of code. In *CGO '11*, pages 289–298, 2011.

[8] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *PLDI '01*, pages 24–34, 2001.

[9] J. Kodumal and A. Aiken. The set constraint/CFL reachability connection in practice. In *PLDI '04*, pages 207–218, 2004.

[10] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *CC '03*, pages 153–169, 2003.

[11] D. Melski and T. Reps. Interconvertbility of set constraints and context-free language reachability. In *PEPM '97*, pages 74–89, 1997.

[12] P. H. Nguyen and J. Xue. Interprocedural side-effect analysis and optimisation in the presence of dynamic class loading. In *ACSC '05*, pages 9–18, 2005.

[13] E. M. Nystrom, H. seok Kim, and W. mei W. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *SAS '04*, pages 165–180, 2004.

[14] T. Reps. Program analysis via graph reachability. In *ILPS '97*, pages 5–19, 1997.

[15] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI '06*, pages 387–400, 2006.

[16] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for Java. In *OOPSLA '05*, pages 59–76, 2005.

[17] Y. Sui, S. Ye, J. Xue, and P.-C. Yew. SPAS: Scalable path-sensitive pointer analysis on full-sparse ssa. In *APLAS '11*, pages 155–171, 2011.

[18] R. Valle-Rai, L. Hendren, V. Sundaresan, P. Lam, and E. Gagnon. Soot - a Java optimization framework. In *CASCON '99*, pages 125–135, 1999.

[19] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *PLDI '95*, pages 1–12, 1995.

[20] G. Xu, A. Rountev, and M. Sridharan. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *ECOOP '09*, pages 98–122, 2009.

[21] J. Xue and P. H. Nguyen. Completeness analysis for incomplete object-oriented programs. In *CC '05*, pages 271–286, 2005.

[22] D. Yan, G. Xu, and A. Rountev. Demand-driven context-sensitive alias analysis for Java. In *ISSTA '11*, pages 155–165, 2011.

[23] M. Yannakakis. Graph theoretic methods in database theory. In *PODS '90*, pages 230–242, 1990.

[24] H. Yu, J. Xue, W. Huo, X. Feng, and Z. Zhang. Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In *CGO '10*, pages 218–229, 2010.

[25] X. Zheng and R. Rugina. Demand-driven alias analysis for C. In *POPL '08*, pages 197–208, 2008.