An Incremental Points-to Analysis with CFL-Reachability

Yi Lu[1], Lei Shang[1], Xinwei Xie[1 2], and Jingling Xue[1]

[1] Programming Languages and Compilers Group
School of Computer Science and Engineering
University of New South Wales, Sydney, NSW 2052, Australia
{ylu,shangl,xinweix,jingling}@cse.unsw.edu.au
[2] School of Computer Science
National University of Defence Technology, Changsha 410073, China

Abstract. Developing scalable and precise points-to analyses is increasingly important for analysing and optimising object-oriented programs where pointers are used pervasively. An incremental analysis for a program updates the existing analysis information after program changes to avoid reanalysing it from scratch. This can be efficiently deployed in software development environments where code changes are often small and frequent. This paper presents an incremental approach for demand-driven context-sensitive points-to analyses based on Context-Free Language (CFL) reachability. By tracing the CFL-reachable paths traversed in computing points-to sets, we can precisely identify and recompute on demand only the points-to sets affected by the program changes made. Combined with a flexible policy for controlling the granularity of traces, our analysis achieves significant speedups with little space overhead over reanalysis from scratch when evaluated with a null dereferencing client using 14 Java benchmarks.

1 Introduction

Points-to analysis is a static program analysis technique to approximate the set of memory locations that may be pointed or referenced by program variables, which is crucial to software testing, debugging, program understanding and optimisation. But performing precise points-to analysis is an expensive activity, even for small programs. Developing scalable and precise points-to analyses is increasingly important for analysis and optimisation of object-oriented programs where pointers are used pervasively.

Points-to analysis has been studied extensively in order to improve its scalability, precision or tradeoffs [15, 17, 33, 34], and continues to attract significant attention [10, 9, 26, 25, 27, 35, 30, 37, 39]. The majority of the previous solutions perform a whole-program points-to analysis to exhaustively compute points-to information for all variables in the program, which is often too resource-intensive in practice, especially for large programs. Some recent efforts have focused on demand-driven points-to analysis [11, 26, 27, 37], which mostly rely on *Context-Free Language (CFL) reachability* [22] to perform only the necessary work for a set of variables queried by a client rather than a whole-program analysis to find the points-to information for all its variables.

Incremental static analysis seeks to efficiently update existing analysis information about an evolving software system without recomputing from scratch [4], allowing the previously computed information to be reused. Incremental analysis is especially important for large projects in a software development environment where it is necessary to maintain a global analysis in the presence of small and frequent edits. Several solutions have been proposed by using incremental elimination [3, 5], restarting iteration [20], a combination of these two techniques [18], timestamp-based backtracing [13], and logic program evaluation [24]. In this paper, we introduce an incremental approach for points-to analyses based on CFL-reachability.

Many program analysis problems can be solved by transforming them into graph reachability problems [23]. In particular, CFL-reachability is an extension of graph reachability. To perform points-to analysis with CFL-reachability, a program is represented by a *Pointer Assignment Graph (PAG)*, a directed graph that records pointer flow in a program. An object is in the points-to set of a variable if there is a reachable path between them in the PAG, which must be labelled with a string in a specified CFL. Such points-to analysis is typically field-sensitive (by matching load/store edges on the same field), context-sensitive (by matching entry/exit edges for the same call site) and heap-sensitive (by distinguishing the same abstract object from different call paths).

Pointer analyses derived from a CFL-reachability formulation achieve very high precision and are efficient for a small number of queries raised in small programs, but they do not scale well to answer many queries for large programs. Existing solutions address the performance issue from several directions, by using refinement [27, 28], (whole-program) pre-analysis [36], ad hoc caching [41], and procedural summarisation [26, 25, 37]. In this paper, we tackle this issue from a different angle. Our goal is to develop an incremental technique for boosting the performance of points-to analysis by reusing previously computed points-to sets.

In this paper, we combine incremental analysis with graph reachability to obtain naturally a trace-based incremental mechanism for points-to analysis, which is effective and simple to implement. The key to incremental analysis lies in approximating dependency information for analysis results. By observing that each points-to query in a CFL-reachability-based analysis is answered by finding the CFL-reachable paths in the PAG from the queried variable to objects, we trace the set of nodes in the traversed paths that the query depends on. Upon code changes, we can precisely identify and recompute on demand only those queries whose traces may overlap with the changes made. Such trace-based falsification minimises the impact of changes on previously computed points-to sets, avoiding unnecessarily falsifying unaffected queries to make them reusable after code changes. Our approach can support efficiently multiple changes with overlapping traces, since multiple changes usually exhibit locality [40].

Precise tracing is costly in space, because it potentially involves thousands of nodes in a PAG for each query, which may render the whole incremental approach impractical, especially for answering many queries in large programs. It is therefore useful to allow tradeoffs between time and space to be made in an incremental analysis. Based on the observation that a large portion of the analysis is performed on Java library code, which is less likely to be changed, we introduce a flexible policy to control the granularities of traces by approximating the variable nodes with their scopes (e.g., methods, classes, etc.). Such trace policies describe different granularity levels used for different parts of the program; they may be specified by programmers as an input to our analysis, or inferred adaptively based on the frequency of code changes. Typically a finer (e.g., variable-level) granularity may be used for the code that is more likely to be changed frequently (e.g., for the classes being developed) to minimise falsification and recomputation required after code changes, while a coarser (e.g., package-level) granularity may be used for the code that is less frequently edited (e.g., for the classes in libraries) to minimise the space required for storing the traces as required. In our experiments, we find that only a small part of code needs to use finer granularities. By using the appropriate granularities for different parts of the programs, we are able to maintain sufficient dependency information for precise falsification with little memory overhead.

In summary, this paper makes the following contributions:

– We propose a trace-based incremental approach for points-to analysis by exploiting graph reachability. To our knowledge, this is the first points-to analysis with CFL-reachability that allows previously computed points-to sets to be reused.
– We introduce a flexible trace policy to approximate traces. Programmers may take advantage of domain knowledge to control their granularities for different parts of the program. We also describe an adaptive technique to automatically infer the policy based on the frequency of changes. Trace policies can significantly reduce the size of traces without unnecessarily increasing the chances of falsification.
– We have implemented our incremental analysis in *Soot-2.5.0*, a Java optimisation and analysis framework, and compared it with a state-of-the-art from-scratch analysis, REFINEPTS, introduced in [27] using a null dereferencing client in the presence of small code changes. For a single deletion of a class/method/statement, our incremental analysis achieves an average speedup of 78.3X/60.1X/3195.4X.

The rest of the paper is organised as follows. We introduce the background information on CFL-reachability and PAGs in Section 2. Section 3 introduces reachability traces by example. Section 4 presents our trace-based incremental analysis, including trace policies. Experimental results are presented and analysed in Section 5 with related work discussed in Section 6, followed by a brief conclusion in Section 7.

2 CFL-Reachability

We introduce the state-of-the-art points-to analysis for Java formulated in terms of CFL-reachability [26–28, 36] which uses Spark's PAGs [17] as the program representation. In Section 2.1, we consider the syntax of PAGs and how to represent a Java example as a PAG. In Section 2.2, we describe the CFL-reachability formulation and show how to answer points-to queries by finding reachable paths in the PAG of our example.

2.1 Program Representation

Points-to analysis for Java is typically flow-insensitive, field-sensitive and context-sensitive (for both method calls and heap abstraction) to balance the precision and efficiency for demand queries. When an analysis is flow-insensitive, control-flow statements are irrelevant and thus ignored.

In its canonical form, a Java program is represented by a directed graph, known as a Pointer Assignment Graph (PAG), which has threes types of nodes: objects, local variables and global variables. The syntax of PAG is given in Fig. 1.

Local variables	x, y	Allocation sites	o
Global variables	g	Call sites	i
Variables	$v ::= x \mid g$	Instance fields	f
Nodes	$n ::= o \mid v$		

$$\text{Edges} \quad e ::= x \xleftarrow{\textsf{new}} o \mid x \xleftarrow{\textsf{assign}} y \mid v \xleftarrow{\textsf{global}} g \mid g \xleftarrow{\textsf{global}} v$$
$$\mid x \xleftarrow{\textsf{ld}(f)} y \mid x \xleftarrow{\textsf{st}(f)} y \mid x \xleftarrow{\textsf{entry}_i} y \mid x \xleftarrow{\textsf{exit}_i} y$$

Fig. 1. Syntax of PAG.

All edges are oriented in the direction of value flow, representing the statements in the program. For example, $x \xleftarrow{\textsf{new}} o$ indicates the flow of o into x (an assignment from an allocation site o to a local variable x). As a result, x points directly to o. An assign edge represents an assignment between local variables (e.g., $x = y$), so x points to whatever y points to. In a global edge, one or both variables are static variables in a class. A ld edge reads an instance field f (e.g., $x = y.f$) while a st edge writes to an instance field f (e.g., $x.f = y$), where x and y are both local variables. An $\textsf{entry}_i$ edge represents a binding of a (local) actual parameter y to its corresponding formal parameter x for a call at the call site i. Similarly, an $\textsf{exit}_i$ edge represents a call return where the (local) return value in y is bound to the local variable x at the call site i.

Fig. 2 gives an example, extending the original example in [27], which provides an abstraction for the Java container pattern. The `AddrBook` class makes use of two vectors. In lines 42–45, an `AddrBook` object created is assigned to `p` and populated with a pair of objects: one with type `String` and the other with type `Integer`. In lines 46 and 47, calling `getName`/`getNum` results in `v1` = `n` and `v2` = `c`. Note that loads and stores to array elements are modeled by collapsing all elements into a special field arr.

For this example, its PAG is shown in Fig. 3. Some notations are in order: (1) o_i denotes the abstract object o created at the allocation site in line i; (2) for temporary variables (e.g., `ret` and `tmp`), the implicit self variable (`this`) and local variables (declared in different scopes), we subscript them with their method names.

2.2 Points-to Analysis with CFL-Reachability

CFL-reachability [22, 38] is an extension of graph reachability that is equivalent to the reachability problem formulated in terms of either recursive state machines [7] or set constraints [14]. Each reachable path in a PAG has a string formed by concatenating in order the labels of edges in the path, where load/store pairs on the same field must be matched (field sensitivity) and entry/exit pairs for the same callsite must be matched (context sensitivity). An object is in the points-to set of a variable if there is a reachable (or *flowsTo*) path from the object to the variable. Two variables may be aliases if there is a reachable path from an object to both of them.

```
 1 class AddrBook{
 2   private Vector names, nums;
 3   AddrBook(){
 4     n = new Vector();
 5     c = new Vector();
 6     this.names = n;
 7     this.nums = c; }
 8   Object getName(Integer i){
 9     n = this.names;
10     c = this.nums;
11     for (int j=0;j<c.count;j++)
12       if (c.get(j)==i)
13         return n.get(j);
14     return null; }
15   Object getNum(String s){
16     n = this.names;
17     c = this.nums;
18     for (int i=0;i<n.count;i++)
19       if (n.get(i)==s)
20         return c.get(i);
21     return null; }
22   void addAddr(Object s, Object i){
23     n = this.names;
24     c = this.nums;
25     n.add(s);
26     c.add(i);
27 }}
28 class Vector{
29   Object[] elems; int count;
30   Vector(){
31     t = new Object[MAXSIZE];
32     this.elems = t; }
33   void add(Object e){
34     t = this.elems;
35     t[count++] = e;    // writes t.arr
36   }
37   Object get(int i){
38     t = this.elems;
39     return t[i];       // reads t.arr
40 }}
41 static void main(String[] args){
42   AddrBook p = new AddrBook();
43   String n = new String("John Smith");
44   Integer c = new Integer(12345);
45   p.addAddr(n,c);
46   String v1 = (String) p.getName(c);
47   Integer v2 = (Integer) p.getNum(n);
48 }
```

(a) original code

```
25     s = new String("Change1"); n.add(s);        // Change 1
26     i = new String("Change2"); c.add(i);        // Change 2
```

(b) code changes

Fig. 2. A Java example.

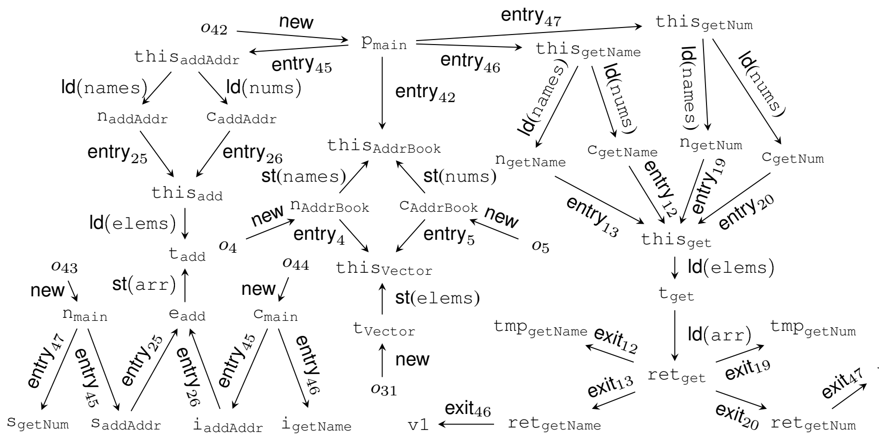

Fig. 3. Complete PAG for the original code.

Field Sensitivity Let us start by considering 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. Then we have the following (regular) grammar for L_{FT}:

$$\textit{flowsTo} \rightarrow \text{new} \; (\; \text{assign})^*$$

If o *flowsTo* v, then 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* [27]. Two variables may be aliases if an object may flow to both of them. Thus, v may point to o flowing into v' if there exists two statements $x.f = v'$ and $v = y.f$, such that the base variables x and y are aliases. So o flows through the two statements with a pair of parentheses (i.e., $\mathsf{st}(f)$ and $\mathsf{ld}(f)$), first into v' and then into v. Therefore, the *flowsTo* production is extended into:

$$\textit{flowsTo} \rightarrow \mathsf{new}\ (\ \mathsf{assign} \mid \mathsf{st}(f)\ \textit{alias}\ \mathsf{ld}(f))^*$$

where x *alias* y means that x and y can 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 ρ can be inverted to obtain its corresponding $\overline{\textit{flowsTo}}$-path $\overline{\rho}$ using inverse edges, and vice versa. For each edge $x \overset{\ell}{\leftarrow} y$ in ρ (where ℓ is the label of the edge), its inverse edge is $y \overset{\overline{\ell}}{\leftarrow} x$ in $\overline{\rho}$. 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, a $\overline{\textit{flowsTo}}$-path represents a *points-to path*. (To avoid cluttering, the inverse edges in a PAG, such as the one given in Fig. 3, are not shown explicitly.) As a result, x *alias* y iff x $\overline{\textit{flowsTo}}$ o *flowsTo* y:

$$\begin{aligned}\textit{alias} &\rightarrow \overline{\textit{flowsTo}}\ \textit{flowsTo}\\ \overline{\textit{flowsTo}} &\rightarrow (\ \overline{\mathsf{assign}} \mid \overline{\mathsf{ld}(f)}\ \textit{alias}\ \overline{\mathsf{st}(f)})^*\ \overline{\mathsf{new}}\end{aligned}$$

Context Sensitivity A context-sensitive analysis requires call entries and exits to be matched, which is solved also as a balanced-parentheses problem [22]. This is done by filtering out *flowsTo*- and $\overline{\textit{flowsTo}}$-paths corresponding to unrealisable paths. A realisable path may not start and end in the same method. So partially balanced parentheses, i.e., a prefix (suffix) with unbalanced closed (open) parentheses, are allowed.

To compute the points-to set of a variable v, we simply solve a CFL-reachability problem for L_{FT} context-sensitively to find the set of allocation sites o such that v is L-reachable from o. The analysis is *fully* context-sensitive not only for method invocation but for heap abstraction (by distinguishing allocation sites with calling contexts).

Example We use the PAG of our example in Fig. 3 to show how to resolve some simple points-to relations via CFL-reachability. Let us see how to discover o_4 as a pointer target for $\mathtt{n}_{\mathtt{addAddr}}$. In Fig. 2, o_{42} flows to $\mathtt{p}_{\mathtt{main}}$, which is the actual parameter passed to the formal parameter $\mathtt{this}_{\mathtt{AddrBook}}$ of constructor `AddrBook` and $\mathtt{this}_{\mathtt{addAddr}}$ of `addAddr`. So $\mathtt{this}_{\mathtt{AddrBook}}$ *alias* $\mathtt{this}_{\mathtt{addAddr}}$. This fact is found in L_{FT} because

$$\mathtt{this}_{\mathtt{AddrBook}}\ \overline{\mathsf{entry}_{42}}\ \mathtt{p}_{\mathtt{main}}\ \overline{\mathsf{new}}\ o_{42}\ \mathsf{new}\ \mathtt{p}_{\mathtt{main}}\ \mathsf{entry}_{45}\ \mathtt{this}_{\mathtt{addAddr}}$$

We then know that o_4 *flowsTo* $\mathtt{n}_{\mathtt{addAddr}}$ since L_{FT} has the *flowsTo*-path:

$$o_4\ \mathsf{new}\ \mathtt{n}_{\mathtt{AddrBook}}\ \mathsf{st}(\mathtt{names})\ \mathtt{this}_{\mathtt{AddrBook}}\ \textit{alias}\ \mathtt{this}_{\mathtt{addAddr}}\ \mathsf{ld}(\mathtt{names})\ \mathtt{n}_{\mathtt{addAddr}}$$

This *flowsTo*-path is a realisable path. So $\mathtt{n}_{\mathtt{addAddr}}$ points to o_4.

3 Tracing CFL-Reachability: An Example

Most points-to analyses only consider fixed programs. We illustrate how we cope with program changes using the example given in Fig. 2. There are two changes made to the original code in Fig. 2(a), in order in line 25 and line 26 as shown in Fig. 2(b). We show

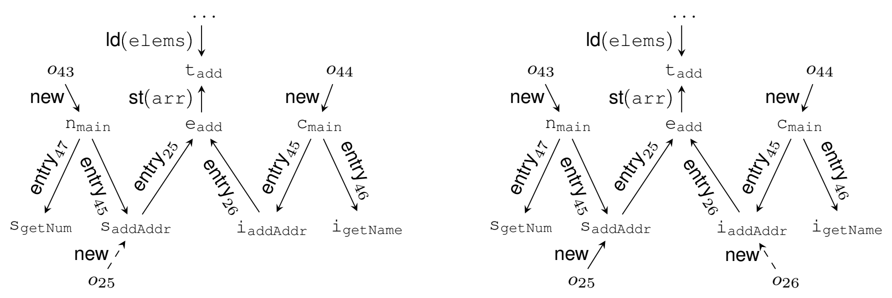

Fig. 4. Partial PAGs after code changes (marked by the dashed edges introduced).

how these changes impact the points-to sets of `v1` and `v2`. Fig. 4 shows the partial PAGs after each change. We have used dashed arrows to indicate newly added edges.

A points-to query is answered by searching for all reachable paths between objects and the queried variable in a PAG. The answer to the points-to query depends on all nodes in the reachable paths traversed. Changes made on these nodes (by either adding or deleting edges connected to them) may falsify the points-to set of the query. The key to incremental analysis lies in tracking such dependent information.

A straightforward way to track precise dependency information is to explicitly maintain a set of variable nodes on which each points-to query depends, as traces. Let us consider the traces for queries on `v1` and `v2` in Fig. 2 and see how they are affected by code changes. By collecting all distinct variable nodes in the reachable path(s) from o_{43} to `v1` in Fig. 3, we get the trace for `v1`: $\{\texttt{v1}, \texttt{ret}_{\texttt{getName}}, \texttt{ret}_{\texttt{get}}, \texttt{t}_{\texttt{get}}, \texttt{this}_{\texttt{get}}, \texttt{n}_{\texttt{getName}}, \texttt{this}_{\texttt{getName}}, \texttt{p}_{\texttt{main}}, \texttt{this}_{\texttt{AddrBook}}, \texttt{n}_{\texttt{AddrBook}}, \texttt{this}_{\texttt{Vector}}, \texttt{t}_{\texttt{Vector}}, \texttt{this}_{\texttt{addAddr}}, \texttt{n}_{\texttt{addAddr}}, \texttt{this}_{\texttt{add}}, \texttt{t}_{\texttt{add}}, \texttt{e}_{\texttt{add}}, \texttt{s}_{\texttt{addAddr}}, \texttt{n}_{\texttt{main}}\}$ and the trace for `v2`: $\{\texttt{v2}, \texttt{ret}_{\texttt{getNum}}, \texttt{ret}_{\texttt{get}}, \texttt{t}_{\texttt{get}}, \texttt{this}_{\texttt{get}}, \texttt{c}_{\texttt{getNum}}, \texttt{this}_{\texttt{getNum}}, \texttt{p}_{\texttt{main}}, \texttt{this}_{\texttt{AddrBook}}, \texttt{c}_{\texttt{AddrBook}}, \texttt{this}_{\texttt{Vector}}, \texttt{t}_{\texttt{Vector}}, \texttt{this}_{\texttt{addAddr}}, \texttt{c}_{\texttt{addAddr}}, \texttt{this}_{\texttt{add}}, \texttt{t}_{\texttt{add}}, \texttt{e}_{\texttt{add}}, \texttt{i}_{\texttt{addAddr}}, \texttt{c}_{\texttt{main}}\}$.

Change 1 adds a new edge to the local variable $\texttt{s}_{\texttt{addAddr}}$. Since the variable is in the trace of `v1`, after the change, the points-to set of `v1` must be falsified and recomputed. However, the trace of `v2` does not contain $\texttt{s}_{\texttt{addAddr}}$. Thus, its points-to set is still valid and reusable. Similarly, change 2 adds a new edge to the local variable $\texttt{i}_{\texttt{addAddr}}$. This falsifies the points-to set of `v2` without affecting `v1` .

Tracing all nodes is costly in space as traces may be large for large programs. Instead of tracking precise dependencies, we approximate the variables in a trace using their scopes, based on a predefined policy. Trace policies control the granularities of traces so that their sizes can be significantly reduced to trade time for space.

Policies are formed by a set of program units (variables or their scopes), which specify what may appear in traces. For example, if the policy for an analysis contains a method name m, when nodes n_1 and n_2 are reached in computing a points-to set such that n_1 and n_2 are contained (defined) in m, m (instead of n_1 and n_2) is tracked in the trace of the points-to set. The default granularity is package-level. For example, if n is in a reachable path but not contained in any program unit in the given policy, then n's package name is used in the trace. This is particularly useful for specifying an appropriate granularity for libraries. We do not have to explicitly include anything from libraries in the policy. They are by default tracked at the coarsest package-level granularity, because they are the least likely to change. For applications being developed in an interactive programming environment, it is natural to use a finer granularity.

Let us now consider how traces and falsifications are enforced by trace policies. We define a sample trace policy for analysing the Java example in Fig. 2: {main, AddrBook.AddrBook, getName, getNum, addAddr}, which uses method-level granularity for the AddrBook class and the main method (they are considered as application code in contrast to library code). Note that Vector is not explicitly mentioned in the policy, since it is considered as part of library code. As a result, the default package-level granularity is used to track the changes on Vector. In Section 4.2, we introduce some forms of shorthand to simplify the specification of policies.

By applying this policy (assuming that the Vector class is defined in package util), the trace for v1 becomes much smaller: {main, getName, AddrBook.AddrBook, addAddr, util} and the trace for v2 is also smaller: {main, getNum, AddrBook.AddrBook, addAddr, util} . Clearly, either Change 1 or Change 2 may falsify both v1 and v2, because we have used a method-level granularity for the changes made in addAddr. It is possible to define a finer-grained policy for the code being changed, but it may not always be possible to anticipate where changes will be made.

Policies can be inferred automatically based on the frequency of code changes in different parts of a program. Typically finer-grained policies may be inferred for frequently edited code and coarser-grained policies for code that is less likely to be modified. Therefore, the impact of changes on the points-to information related to the frequently changed code may be kept to a minimum.

Let us show how to infer policies adaptively. The initial policy is empty (or supplied by the programmer) and the traces of v1 and v2 are {my_package, util}, assuming that AddrBook and main are defined in the my_package package. After Change 1 at line 25, the policy is adaptively changed to $\{\mathtt{this}_{\mathrm{addAddr}}, \mathtt{s}_{\mathrm{addAddr}}, \mathtt{i}_{\mathrm{addAddr}}, \mathtt{n}_{\mathrm{addAddr}}, \mathtt{c}_{\mathrm{addAddr}},$ getName, AddrBook.AddrBook, getNum, addAddr, AddrBook, main} by adding finer-grained program units into the policy based on the change made.

Since code changes often exhibit locality, we choose a simple heuristic to reduce the chance of falsification for units that are closely related to a certain change. When a program unit is involved in a change, we add all units directly defined in its enclosing scopes. For example, Change 1 affects variable $\mathtt{s}_{\mathtt{addAddr}}$ and transitively all its enclosing scopes: method addAddr, class AddrBook and package my_package. Therefore, all variables defined in addAddr, all methods defined in AddrBook, and all classes defined in my_package are added into the new policy.

After Change 1 at line 25, both v1 and v2 are conservatively falsified, because the change overlaps with my_package in both traces. After recomputing the points-to sets for v1 and v2 using the new policy, the new trace of v1 is $\{\mathtt{s}_{\mathrm{addAddr}}, \mathtt{this}_{\mathrm{addAddr}}, \mathtt{n}_{\mathrm{addAddr}},$ AddrBook.AddrBook, getName, main, util} , and the new trace of v2 is {AddrBook.AddrBook, getNum, $\mathtt{this}_{\mathrm{addAddr}}, \mathtt{c}_{\mathrm{addAddr}}, \mathtt{i}_{\mathrm{addAddr}},$ main, util} .

After Change 2 at line 26, an incremental analysis is used again. This time only the points-to set of v2 is falsified, because $\mathtt{i}_{\mathtt{addAddr}}$ does not overlap with the trace of v1. The previous points-to set for v2 before the change was $\{o_{44}\}$. We knew that the typecasting at line 47 was safe because the type of o_{44} was Integer; we could omit a runtime check for this cast. After recomputing the query, the points-set of `v2` becomes $\{o_{44}, o_{26}\}$, where o_{26} is introduced by Change 2. Since o_{26} is a `String`, we know that the typecasting at line 47 may no longer be safe.

4 Incremental Analysis with Reachability Traces

In this section, we describe our incremental analysis formally using inference rules [11, 28, 24]. Our goal is to incrementalise the points-to analysis based on CFL-reachability.

In Section 4.1, we first introduce a simple form of incremental points-to analysis based on reachability traces, where all nodes in the reachable paths traversed in computing the points-to sets are traced. In Section 4.2, we then introduce a space-efficient analysis by approximating traces using policies. Finally, in Section 4.3, we show how to infer trace policies adaptively with each incremental analysis.

Our incremental analysis proceeds in two phases: initial phase and incremental phase. The initial phase initialises the whole analysis by answering all queries from scratch, and the answers (points-to sets) are cached for reuse. This occurs only when a new program is analysed or after major program changes, where it is necessary to reinstall the whole analysis. Unlike the initial phase, the incremental phase performs falsification in addition to points-to analysis. This occurs after small changes and only recomputes a small number of cached answers that are falsified by the changes made.

4.1 Points-to Analysis with Reachability Traces

We have developed our approach based on insights gained from formulating points-to analysis as a graph reachability problem. Our CFL-reachability-based analysis computes both points-to information and traces. The additional syntax is given in Fig. 5.

Contexts	$k ::= \varnothing \mid k{:}i$
Traces/Changes	$\tau, \Delta ::= \varnothing \mid \{\mu\} \mid \tau \cup \tau$
Program units	$\mu ::= v$
Points-to sets	$\sigma ::= \varnothing \mid \{o\} \mid \sigma \cup \sigma$
Stores	$\Sigma ::= \varnothing \mid \Sigma, v \mapsto (\sigma, \tau)$

Fig. 5. Additional syntax for points-to analysis.

Contexts represent how method calls are made. A calling context k is a finite stack of call sites, whose order is significant. Traces track nodes traversed in reachable paths in computing points-to sets. A trace τ is a set of variable nodes (we do not track object nodes), whose order is not significant. A points-to set σ contains a set of objects cached in a store Σ, which maps variables to their points-to sets and traces.

In addition to performing the standard CFL-reachability-based points-to analysis in the PAG of a program, we maintain the traces along the search. Our points-to analysis is described in Fig. 6 by a set of inference rules in the form of:

$$n, k \xRightarrow{\tau} n', k'$$

$$\frac{x \xrightarrow{\overline{\mathsf{new}}} o}{x, k \overset{\{x\}}{\Longrightarrow} o, k}\ \text{(new)} \qquad \frac{x \xrightarrow{\overline{\mathsf{entry}_i}} y}{x, \varnothing \overset{\{x\}}{\Longrightarrow} y, \varnothing}\ \text{(entry-}\varnothing\text{)}$$

$$\frac{x \xrightarrow{\overline{\mathsf{assign}}} y}{x, k \overset{\{x\}}{\Longrightarrow} y, k}\ \text{(assign)} \qquad \frac{x \xrightarrow{\overline{\mathsf{exit}_i}} y}{x, k \overset{\{x\}}{\Longrightarrow} y, k{:}i}\ \text{(exit)}$$

$$\frac{v \xrightarrow{\overline{\mathsf{global}}} g}{v, k \overset{\{v\}}{\Longrightarrow} g, \varnothing}\ \text{(global-r)}$$

$$\frac{x \xrightarrow{\overline{\mathsf{ld}(f)}} x' \qquad y' \xrightarrow{\overline{\mathsf{st}(f)}} y \qquad x', k \overset{\tau}{\Longrightarrow} o, k'' \qquad y', k' \overset{\tau'}{\Longleftarrow} o, k''}{x, k \overset{\{x\} \cup \tau \cup \tau'}{\Longrightarrow} y, k'}\ \text{(field)}$$

$$\frac{g \xrightarrow{\overline{\mathsf{global}}} v}{g, \varnothing \overset{\{g\}}{\Longrightarrow} v, \varnothing}\ \text{(global-l)}$$

$$\frac{x \xrightarrow{\overline{\mathsf{entry}_i}} y}{x, k{:}i \overset{\{x\}}{\Longrightarrow} y, k}\ \text{(entry)} \qquad \frac{n, k \overset{\tau}{\Longrightarrow} n'', k'' \qquad n'', k'' \overset{\tau'}{\Longrightarrow} n', k'}{n, k \overset{\tau \cup \tau'}{\Longrightarrow} n', k'}\ \text{(transitivity)}$$

Fig. 6. Points-to analysis with traces.

which follow the $\overline{\textit{flowsTo}}$ paths, i.e., the *flowsTo* paths in the opposite direction in a PAG. Each edge in a $\overline{\textit{flowsTo}}$ path is translated into one or more inference rules. For example, node n in context k can be reached by node n' in context k' via a set of nodes in trace τ. Traces are computed by tracking nodes along the traversal. To save space, object nodes o tracked by (new) do not need to appear in a trace as they can be uniquely identified by their corresponding left-hand side variables x that appear in the trace.

Global variables are context-insensitive (as our analysis is flow-insensitive). Thus, (global-r) and (global-r) skip the sequence of calls and returns between the reads and writes of a global variable. (entry) and (exit) achieve context sensitivity for method invocation by matching call entries and exits. (entry-$\varnothing$) allows for partially balanced parentheses. (field) achieves field sensitivity for field accesses (reads and writes) by matching field loads and stores on field f, only if x' and y' may be aliases (when there is an object o that may be pointed by x' and may flow to y'). In this rule, $\Longleftarrow$ denotes the flows-to analysis which is analogous to its inverse points-to analysis (by making inferences based on traversing the *flowsTo* paths in a PAG) and thus omitted.

Given a points-to query for variable v, we find its point-to set, denoted as $Pts(v)$, by deriving all possible reachable paths ending with some objects:

$$Pts(v) = \{(o, \tau) \mid v, \varnothing \overset{\tau}{\Longrightarrow} o, k\} \qquad \text{(pointsto)}$$

where o is a pointed-to object in context k and τ is the trace for computing it.

Initial Phase During this phase, all queries are answered by computing points-to sets from scratch. The initial analysis $\mathit{Initialise}$ takes a set of queried variables $(v_{1..n})$ as input, and computes and caches each variable's points-to set and trace in the store Σ_n.

$$\frac{\begin{array}{c}\Sigma_0 = \varnothing \\ \forall i \in 1..n \quad \cdot \quad Pts(v_i) = (o, \tau)_{1..m} \qquad \Sigma_i = \Sigma_{i-1}, v_i \mapsto (\bigcup o_{1..m}, \bigcup \tau_{1..m})\end{array}}{\mathit{Initialise}(v_{1..n}) = \Sigma_n} \qquad \text{(initialisation)}$$

Incremental Phase Our incremental technique is based on the observation that if a points-to set becomes invalid after a code change, then some part of its trace must be involved in the change. In this phase, the incremental analysis $\mathit{Increment}$ takes the changes Δ (represented by a set of program units that are affected by either additions or deletions of program constructs) and the points-to store Σ_0 from the previous analysis as input, and returns an update-to-date store Σ_n, where only the points-to sets affected by the changes (whose traces τ_i overlap the changes Δ) are recomputed.

$$\frac{\begin{array}{c}\Sigma_0 = (v \mapsto (\sigma, \tau))_{1..n} \qquad \forall i \in 1..n \quad \cdot \\ \mathit{if}\ \ \Delta \perp \tau_i\ \ \mathit{then}\ \begin{cases} \mathit{Pts}(v_i) = (o, \tau')_{1..m} \\ \Sigma_i = \Sigma_{i-1}[v_i \mapsto (\bigcup o_{1..m}, \bigcup \tau'_{1..m})] \end{cases} \\ \mathit{else}\ \ \Sigma_i = \Sigma_{i-1} \end{array}}{\mathit{Increment}(\Delta, \Sigma_0) = \Sigma_n} \quad \text{(increment)}$$

We define the inference rules for determining if the changes overlap with a trace:

$$\frac{\mu \in \tau \qquad \mu' \in \tau' \qquad \{\mu\} \perp \{\mu'\}}{\tau \perp \tau'} \ \text{(overlap-trace)} \qquad\qquad \tau \perp \tau \quad \text{(overlap-reflectivity)}$$

Here, traces or changes are only sets of variables, as a program unit μ can only be a variable. In the next section, we will provide a more flexible model to handle different types of program units, such as methods, classes and packages.

4.2 Saving Space with Trace Policies

Trace policies control the granularities of traces in order to trade analysis time for memory usage. In Fig. 7, we introduce method, class and package names into the syntax of program units μ, which form traces τ (and changes Δ). Policies are also formed by a set of program units, which specify what program units may appear in traces.

Method names	m		
Class names	c	Program units	$\mu ::= \cdots \mid m \mid c \mid p$
Package names	p	Policies	$\Gamma ::= \varnothing \mid \mu \mid \Gamma \cup \Gamma$

Fig. 7. Syntax of trace policies.

Policies may be defined by programmers. Writing down all program units to be tracked in traces may not be practical. To simplify the specification of policies, we introduce some forms of shorthand, formally defined by the syntactical equivalence:

$$\{\mu : \mathsf{variable}\} \equiv \{v \mid v \trianglelefteq \mu\} \quad \text{(policy-variable)}$$
$$\{\mu : \mathsf{method}\} \equiv \{m \mid m \trianglelefteq \mu\} \quad \text{(policy-method)}$$
$$\{\mu : \mathsf{class}\} \equiv \{c \mid c \trianglelefteq \mu\} \quad \text{(policy-class)}$$

Often programmers may simply specify a single line $\mathit{my\_package}$:method in the policy to indicate that $\mathit{my\_package}$ is the package being developed and request the method-granularity to be used. The shorthand essentially includes all methods contained in $\mathit{my\_package}$. Any other code changes are tracked at package-level, which is the default (avoiding a need for a shorthand).

The containment relation between program units is reflective and transitive. We capture it using the structure of a Java program with a few mappings. P maps a package name to the set of names of all classes defined in the package. C maps a class name to the set of names of all methods and global variables defined in the class. M maps a method name to the set of names of all local variables defined in the method. Given P, C and M, we can easily find containment relations between each pair of program units:

$$\frac{x \in M(m)}{x \trianglelefteq m} \quad \text{(contain-local)} \qquad \frac{c \in P(p)}{c \trianglelefteq p} \quad \text{(contain-class)}$$

$$\frac{g \in C(c)}{g \trianglelefteq c} \quad \text{(contain-global)} \qquad \frac{\mu \trianglelefteq \mu'' \qquad \mu'' \trianglelefteq \mu'}{\mu \trianglelefteq \mu'} \quad \text{(contain-transitivity)}$$

$$\frac{m \in C(c)}{m \trianglelefteq c} \quad \text{(contain-method)} \qquad \mu \trianglelefteq \mu \quad \text{(contain-reflectivity)}$$

Now we define the rules for approximating a trace according to a given policy:

$$\frac{\forall i \in 1..n \quad \cdot \quad \mu_i = \mathit{Approx}(v_i, \Gamma)}{\mathit{Approx}(v_{1..n}, \Gamma) = \bigcup \mu_{1..n}} \quad \text{(approx-trace)}$$

$$\frac{v \trianglelefteq p \qquad \mathit{if}\ v \trianglelefteq \mu\ \mathit{then}\ \mu \notin \Gamma}{\mathit{Approx}(v, \Gamma) = p} \quad \text{(approx-default)}$$

$$\frac{\begin{array}{c} v \trianglelefteq \mu \qquad \mu \in \Gamma \\ \forall \mu' \in \Gamma \quad \cdot \quad \mathit{if}\ v \trianglelefteq \mu'\ \mathit{then}\ \mu \trianglelefteq \mu' \end{array}}{\mathit{Approx}(v, \Gamma) = \mu} \quad \text{(approx-contain)}$$

In (approx-default), if no enclosing scope of v is defined in the policy, then its package is tracked by default. (approx-contain) only finds and tracks the smallest enclosing scope in the policy. For example, if we find that both the method name and class name of v are in the policy, we will then use the method name as its granularity.

Initial Phase The initial analysis is slightly modified to approximate the traces before they are stored, according to the given policy as an input:

$$\frac{\begin{array}{c} \Sigma_0 = \varnothing \\ \forall i \in 1..n \quad \cdot \quad \mathit{Pts}(v_i) = (o, \tau)_{1..m} \\ \Sigma_i = \Sigma_{i-1}, v_i \mapsto (\bigcup o_{1..m}, \mathit{Approx}(\bigcup \tau_{1..m}, \Gamma)) \end{array}}{\mathit{Initialise2}(v_{1..n}, \Gamma) = \Sigma_n} \quad \text{(initialisation-2)}$$

Incremental Phase The incremental analysis is also slightly changed to approximate the traces when recomputing points-to sets:

$$\frac{\begin{array}{c} \Sigma_0 = (v \mapsto (\sigma, \tau))_{1..n} \qquad \forall i \in 1..n \quad \cdot \\ \mathit{if}\ \ \Delta \perp \tau_i\ \ \mathit{then}\ \begin{cases} \mathit{Pts}(v_i) = (o, \tau')_{1..m} \\ \Sigma_i = \Sigma_{i-1}[v_i \mapsto (\bigcup o_{1..m}, \mathit{Approx}(\bigcup \tau'_{1..m}, \Gamma))] \end{cases} \\ \mathit{else}\ \ \Sigma_i = \Sigma_{i-1} \end{array}}{\mathit{Increment2}(\Delta, \Sigma_0, \Gamma) = \Sigma_n} \quad \text{(increment-2)}$$

We have just extended the syntax of program units in the traces and changes so that we can now directly represent additions/deletions of not only statements but also, for example, methods or classes. We now need to extend the rules for checking overlap among traces/changes. An overlap relation is reflective and symmetric:

$$\frac{\mu \trianglelefteq \mu'}{\{\mu\} \perp \{\mu'\}} \quad \text{(overlap-contain)} \qquad \frac{\tau' \perp \tau}{\tau \perp \tau'} \quad \text{(overlap-symmetry)}$$

4.3 Adaptive Inference of Trace Policies

In order to specify a trace policy, we need to anticipate where changes will be made, which may not always be possible. We describe how to gradually refine trace policies from each incremental analysis, allowing policies to be inferred automatically based on the frequency of changes in different parts of a program. Therefore, the impact of changes on the existing points-to information related to the frequently changed code is minimised.

Initial Phase The trace policy for the initial analysis is either empty or supplied by the programmer, which can be set up by reusing $\mathit{Initialise2}$ from Section 4.2.

Incremental Phase $\mathit{Increment3}$ refines the trace policy by adding finer-grained program units into it. This incremental analysis reuses $\mathit{Increment2}$ after adapting the policy to the changes, and returns the refined policy as output:

$$\frac{\mathit{Adapt}(\Delta) = \Gamma' \qquad \mathit{Increment2}(\Delta, \Sigma, \Gamma \cup \Gamma') = \Sigma'}{\mathit{Increment3}(\Delta, \Sigma, \Gamma) = \Sigma', \Gamma \cup \Gamma'} \quad \text{(increment-3)}$$

The following adaption rules compute finer-grained program units to be added into the policy, based on the type of changes made:

$$\mathit{Adapt}(\varnothing) = \varnothing \quad \text{(adapt-}\varnothing\text{)}$$

$$\mathit{Adapt}(\{\mu\} \cup \Delta) = \mathit{Adapt}(\{\mu\}) \cup \mathit{Adapt}(\Delta) \quad \text{(adapt-changes)}$$

$$\frac{x \trianglelefteq m}{\mathit{Adapt}(\{x\}) = \{y \mid y \trianglelefteq m\} \cup \mathit{Adapt}(\{m\})} \quad \text{(adapt-local)}$$

$$\frac{g \trianglelefteq c}{\mathit{Adapt}(\{g\}) = \{g' \mid g' \trianglelefteq c\} \cup \mathit{Adapt}(\{c\})} \quad \text{(adapt-global)}$$

$$\frac{m \trianglelefteq c}{\mathit{Adapt}(\{m\}) = \{m' \mid m' \trianglelefteq c\} \cup \mathit{Adapt}(\{c\})} \quad \text{(adapt-method)}$$

$$\frac{c \trianglelefteq p}{\mathit{Adapt}(\{c\}) = \{c' \mid c' \trianglelefteq p\} \cup \mathit{Adapt}(\{p\})} \quad \text{(adapt-class)}$$

$$\mathit{Adapt}(\{p\}) = \varnothing \quad \text{(adapt-package)}$$

If a local variable x is changed in (adapt-local), we add all local variables in its method into the policy, and then adapt the policy to the method changed. In general, we add all programs units that are directly defined in the enclosing scopes of x. The last rule (adapt-package) adapts nothing as package-level is the default granularity.

Benchmark	Whole Program			Application Code			#Queries
	#Classes	#Methods	#Statements	#Classes	#Methods	#Statements	
compress	5262	50667	372268	23	175	2989	443
jess	5402	51318	382460	161	798	13099	2064
db	5254	50667	372327	15	175	3035	239
javac	5422	51803	395661	183	1300	26238	5844
mpegaudio	5302	50944	384133	63	410	14869	7644
mtrt	5275	50799	374981	36	304	5714	911
jack	5307	50948	381756	68	443	12486	3296
avrora	2858	24412	197754	549	3194	42946	1413
batik	6827	60013	507723	1114	7356	125770	3574
fop	8441	74894	538179	978	7055	147677	10739
lusearch	2457	23113	190279	220	1979	32124	4053
sunflow	5508	52238	410396	170	1469	35267	1552
tradebeans	9272	83384	533529	909	6787	106480	4353
xalan	3053	28183	258840	618	6253	103348	2093

Table 1. Benchmark statistics. "Whole Program" includes the reachable parts of the Java libraries and "Application Code" does not. The last column gives the number of queries issued.

5 Experimental Evaluation

We evaluate our incremental analysis using a null dereferencing client, `NullDeref`. We compare our analysis with a state-of-the-art from-scratch analysis, REFINEPTS, from [27] using 14 Java programs, selected from the Dacapo and SPECjvm98 benchmark suites, given in Table 1. In the presence of small code changes targeted by this work, our incremental analysis is significantly faster (by at least one order of magnitude) than REFINEPTS when tracing application code at different granularity levels.

5.1 Implementation

We have implemented our incremental analysis and `NullDeref` in the Soot 2.5.0 [32] and Spark [17] framework, and conducted our experiments using the Sun JDK 1.6.0_26 libraries. REFINEPTS is publicly available in the same framework. Unmodeled native methods and reflection calls [19] are handled conservatively using Tamiflex [2]. The on-the-fly call graph of the program is constructed so that a *context-sensitive* call graph is always maintained for a program during the CFL-reachability computation.

5.2 Methodology

We have conducted our experiments on a machine consisting of Intel Xeon 2.27GHz processors (4 cores) with 24 GB memory, running Ubuntu Linux operating system (kernel version 2.6.38). Although the system has multi-cores, each analysis algorithm is single-threaded. Table 1 gives some statistics about the benchmarks used. Columns 2–4 show the number of classes, methods and statements in each program. Columns 5–7 are similar except the Java libraries are excluded. It can be seen that the application code is usually a small part of a Java program, making it suitable to be analysed with different trace policies depending on the nature of program changes made.

`NullDeref` detects null pointer violations, demanding high precision from points-to analysis. Since this client issues a large number of queries, it is suitable to show the affected and unaffected queries after a program change. The last column in Table 1 gives the number of queries issued by the client in a program.

In this paper, we consider changes to the program in terms of node additions and deletions to its program representation (i.e. PAG). To evaluate our incremental analysis, we have selected three different levels of code changes: class, method and statement. Our experiments are conducted by randomly deleting a class/method/statement in the program being analysed, as in [40]. We handle a class-level code change as a set of multiple method-level changes except that we must also handle the changes related to the fields in a changed class. When a field is deleted from a class, all edges related to the field are removed. When a field is added to the class (without statement additions), the PAG needs not to be updated. We have adopted this approach because it is reasonably simple to implement, which enables us to collect data on many potential changes across many programs. We find, in practice, that many code changes do not cause changes to the points-to information; however such code changes are excluded in our experiments.

Traditional points-to analyses like REFINEPTS, which are not designed to accommodate program changes, must recompute points-to information upon a code change. We compare the incremental analysis time, which includes the times on falsification and query processing, with the from-scratch analysis time, which includes the times on PAG construction and query processing. We repeated each experiment 20 times using randomly generated changes and reported the average of the 20 runs. Below we describe and analyse two sets of experiments depending on the granularities used for tracing the application code of a program. In both cases, the library code of a program is traced at package-level since it is unlikely to be modified.

Optimising for Analysis Time. We show the best speedups of our analysis over a from-scratch analysis by tracing application code at variable-level. Our analysis is significantly faster than REFINEPTS and remains so even under a stress test.

Trading Time for Space. We show that our analysis remains to be at least one order of magnitude faster even if we trace application code at method-level or class-level.

At this stage, we do not have results for the scenario when our analysis uses trace policies adaptively, because, unfortunately, we do not have enough change history data to obtain statistically significant results. However, its performance is expected to lie between the two scenarios studied here.

5.3 Optimising for Analysis Time

We consider code changes comprising a single deletion of a class or method or statement. The situation for adding a class or method or statement is similar.

We have compared the analysis times in Table 2 for REFINEPTS (Columns 2–4) and our incremental analysis (Columns 5–7). The execution times are all in seconds. For each program, there are three level of changes: deleting a class (denoted as "del c"), deleting a method (denoted as "del m") and deleting a statement (denoted as "del s"). For REFINEPTS, "PAG" is the time elapsed on constructing the PAG and "QT" denotes the time spent on recomputing all the issued queries. For our incremental analysis, "Falsification" is the time spent on the falsification process and "QT2" is a fraction of "QT" spent on recomputing the affected queries.

		RefinePTS			Incremental Analysis		
		PAG	QT	Total	Falsification	QT2	Total
`compress`	del c	118.8	11.0	129.8	0.011	0.3	0.3
	del m	119.4	6.9	126.3	0.001	0.6	0.6
	del s	118.9	6.5	125.4	0.000	0.09	0.09
`jess`	del c	125.9	157.7	283.6	0.013	70.3	70.3
	del m	122.1	156.0	278.1	0.002	21.8	21.8
	del s	122.1	155.8	277.9	0.001	0.06	0.06
`db`	del c	118.8	12.2	131.0	0.007	0.4	0.4
	del m	119.1	12.2	131.3	0.001	0.5	0.5
	del s	120.2	12.4	132.6	0.000	0.01	0.01
`javac`	del c	125.4	223.4	348.8	0.032	45.5	45.5
	del m	124.8	224.0	348.8	0.006	21.5	21.5
	del s	125.1	226.5	351.6	0.002	4.93	4.93
`mpegaudio`	del c	124.7	27.0	151.7	0.040	8.4	8.4
	del m	121.5	31.0	152.4	0.003	6.5	6.5
	del s	120.8	29.2	150.0	0.001	0.09	0.09
`mtrt`	del c	120.0	28.6	148.6	0.014	2.9	2.9
	del m	118.4	27.2	145.5	0.001	2.4	2.4
	del s	119.3	25.1	144.4	0.001	0.32	0.32
`jack`	del c	118.2	31.0	149.2	0.026	2.5	2.5
	del m	118.4	31.6	150.0	0.001	2.1	2.1
	del s	115.3	27.4	142.7	0.000	0.49	0.49
`avrora`	del c	38.9	15.1	54.0	0.009	1.3	1.3
	del m	37.9	16.8	54.7	0.001	1.6	1.6
	del s	38.7	15.4	54.1	0.001	0.14	0.14
`batik`	del c	141.7	148.9	290.6	0.014	7.3	7.3
	del m	137.4	145.9	283.3	0.003	7.5	7.5
	del s	138.9	141.2	280.1	0.003	0.04	0.04
`fop`	del c	192.0	372.5	564.5	0.065	134.7	134.7
	del m	191.6	378.4	569.9	0.006	28.7	28.7
	del s	190.4	366.4	556.8	0.001	0.11	0.12
`lusearch`	del c	38.0	59.7	97.7	0.010	4.6	4.6
	del m	44.4	63.1	107.5	0.002	4.6	4.6
	del s	38.8	61.8	100.6	0.000	2.03	2.03
`sunflow`	del c	123.3	32.3	155.6	0.018	3.4	3.4
	del m	130.6	28.2	158.7	0.002	4.4	4.4
	del s	126.7	31.0	157.7	0.002	0.48	0.49
`tradebeans`	del c	210.1	256.0	466.1	0.068	23.5	23.6
	del m	214.1	255.3	469.4	0.023	36.5	36.6
	del s	211.4	246.1	457.5	0.001	5.91	5.91
`xalan`	del c	39.2	20.6	59.8	0.009	1.9	1.9
	del m	38.8	20.6	59.4	0.002	1.9	1.9
	del s	36.9	20.3	57.2	0.002	0.02	0.02

Table 2. Analysis times of `NullDeref` in seconds for deleting a class, method or statement.

Our incremental analysis is much faster for all the benchmarks under three different levels of code changes. The average speedups range from 4X to a factor reaching several thousands. This is also true even if only the query time alone is used as a reference, since QT2 is a small fraction of QT. In addition, the falsification process is very fast and negligible relative to QT2. For a single deletion of a class/method/statement, the average speedup is 78.3X/60.1X/3195.4X.

As the library code of a program is traced at package-level, our analysis consumes only 11 MB more memory than RefinePTS in the worst case.

Our incremental analysis is designed to handle small and frequent code changes. Nevertheless, we have stress-tested it with some major changes, involving a deletion of 100 randomly selected methods in a program, as shown in Table 3. While the percentage of valid queries is smaller than the case when only small changes are made, our analysis still outperforms RefinePTS by 1.8X on average.

Our incremental analysis is developed to avoid recomputing unaffected queries after program changes. To understand the sources of performance gains, we have plotted the percentage of unaffected queries, including the “major” changes (with 100 methods deleted) in Fig. 8. On average, 99.1% of the queries are unaffected after a statement deletion. The percentage becomes 93.1% (91.9%) when a method (class) is deleted, respectively. In the case of the major changes, only 33.4% queries are unaffected. Note

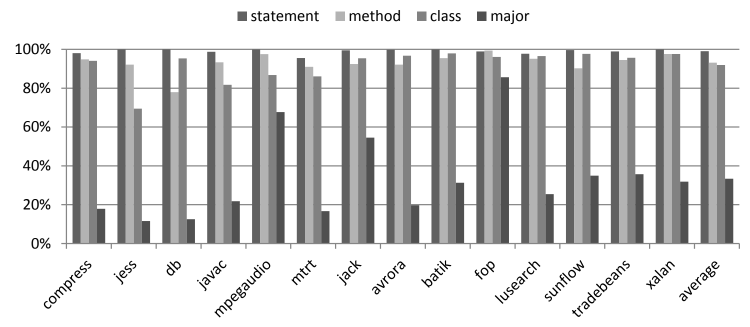

Fig. 8. Percentage of unaffected queries after program changes.

Benchmark	Falsification	QT	#Unaffected Queries (%)	Speedup over REFINEPTS
`compress`	0.020	1.743	17.9	6.1
`jess`	0.043	141.896	11.6	1.1
`db`	0.024	3.611	12.5	3.3
`javac`	0.156	193.899	21.8	1.2
`mpegaudio`	0.095	19.911	67.7	1.3
`mtrt`	0.024	14.576	16.7	1.8
`jack`	0.030	15.364	54.5	1.8
`avrora`	0.040	12.326	19.7	1.4
`batik`	0.041	115.812	31.3	1.2
`fop`	0.531	327.090	85.7	1.2
`lusearch`	0.044	43.284	25.4	1.4
`sunflow`	0.022	18.925	35.0	1.7
`tradebeans`	0.118	212.521	35.7	1.2
`xalan`	0.052	16.826	31.9	1.4
`average`	0.052	81.270	33.4	1.8

Table 3. Stress testing of our analysis with "major" changes (deleting 100 methods).

that neither "method" nor "class" is consistently better than the other in terms of the percentage of affected queries. This may be due to the randomness of our experiments.

5.4 Trading Time for Space

For large programs, tracing the application code of a program at variable-level can be space-prohibitive. Our analysis allows it to be traced at coarser granularities to trade off analysis time for memory usage. As shown in Fig. 9 for a single method deletion, the average trace size (measured in terms of PAG nodes) per query increases as the trace policy becomes coarser. The percentage of unaffected queries for variable-level, method-level and class-level are 93.1%, 87.4% and 74.3%, respectively, on average. As a result, our analysis becomes slower but remains to be at least one order of magnitude faster than a from-scratch analysis. As discussed earlier, our analysis is 60.1X faster than REFINEPTS at variable-level. Its performance speedups has only dropped now to 24.2X and 18.0X at method-level and class-level, respectively.

At the two coarser trace policies, the largest analysis time increases are observed at `mtrt`, which takes 2.438 secs at variable-level but now 9.232 secs at method-level and 13.437 secs at class-level. The speedup of our analysis over REFINEPTS has dropped from 59.7X at variable-level to 15.8X at method-level and 10.X at class-level.

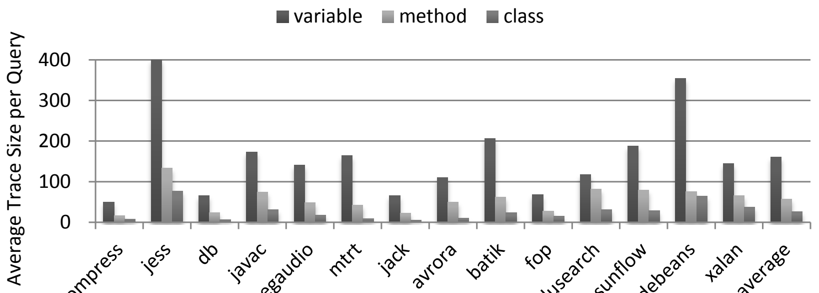

Fig. 9. Trace sizes at three different granularities for a single method deletion.

6 Related Work

In recent years, there has been a large body of research devoted to points-to analysis. We restrict our discussion to three related areas: context-sensitive points-to analysis, incremental analysis and change impact analysis. As demonstrated via a null dereferencing client in our experiments, context sensitivity is needed for Java because many queries issued will not be positively answered otherwise.

Whole-program points-to analysis exhaustively computes points-to information for all its variables, which achieves context sensitivity by cloning [33] or summarisation [12, 34, 39, 31, 29]. Demand-driven points-to analysis [11] reduces the cost of analysis by only computing points-to information that is needed by its client analysis or optimisation. The state-of-the-art algorithms for Java [27, 26, 36] and C [41] are formulated in terms of CFL-reachability initially introduced in [23]. Given a CFL-reachability formulation, demand-driven analyses answer points-to queries as described in Section 2.

Pointer analyses based on CFL-reachability are precise, but they do not scale well to answer many queries for large programs. Sridharan et al. [28, 27] proposed a refinement-based analysis to give an initial approximation and then gradually refine it until the client is satisfied. This strategy is useful for clients that can be satisfied early enough. Xu et al. [36] used an imprecise but cheap pre-analysis to find non-aliasing pairs to reduce redundancy in the subsequent points-to analysis. Zheng and Rugina [41] described a memory alias CFL-reachability formulation, answering alias queries without computing the complete points-to sets. Shang et al. [26] proposed a technique to summarise local points-to relations within a method. Such procedural CFL-reachability summaries may be reused later by the points-to analysis in the same or different calling contexts. In [25], they have also reported preliminary experience of using this technique to summarise the whole program and allow each procedural summary to be updated independently in response to edits from an IDE, achieving a limited form of incrementality. However, it does not allow points-to information to be reused. Therefore, points-to queries are always answered by recomputing from scratch. In contrast, our trace-based incremental algorithm presented in this paper allows previously computed points-to results to be reused, by recomputing only the queries that are falsified by code changes. Our technique is orthogonal to previous ones for improving the scalability of points-to analysis based on CFL-reachability. It may be possible to use our algorithm in conjunction with other techniques such as pre-analysis and procedural summarisation.

Many incremental algorithm have been developed for data-flow analysis problems. Some incremental analyses use the elimination method [3, 5], some are based on the technique of restarting iterations [20] and some are hybrids of the two techniques [18]. A comparison of incremental iterative algorithm can be found in [4].

Incremental points-to analysis has been considered for C programs. Yur et al. [40] introduced an incremental approximation of their previous flow- and context-sensitive alias analysis [15] for C, by falsifying the aliases affected by the changed statements. Their algorithm handles addition/deletion of one single statement, achieving a 6-fold speedup for programs with 1 – 25K LOC. Their analysis is less precise than the reanalysis from scratch (with a solution agreement on 75% of tests on average). In contrast, our incremental algorithm produces exactly the same results as their full-analysis counterpart, and naturally handles multiple changes efficiently.

Kodumal and Aiken [13] considered for a limited form of incremental analysis via backtracking in their Banshee toolkit, which allows constraint systems to be rolled back to any previous state for a code change and reanalyses the program from that point forward. Their coarse-grained analysis is fast but imprecise due to its lack of support for context sensitivity. Saha and Ramakrishnan [24] extended [11], also for C, based on techniques for incremental evaluation of logic programs. When context sensitivity is considered, their analysis is slow, by consuming 50 – 73% of the from-scratch time.

Change impact analysis determines the effects of code changes to support the planning, implementation and validation of code changes in software evolution and maintenance. A taxonomy for impact analysis can be found in [16]. Recent approaches [1, 6, 8, 21] rely on slicing, dependence analysis, dynamic tracing and history mining. In general, impact analysis requires fast and precise points-to information to be effective, which may benefit from our incremental points-to analysis.

7 Conclusion

Incremental points-to analysis is important in large projects where it is necessary to maintain a global analysis in the presence of small edits. We have described an incremental approach via tracing graph reachability, a mechanism that is efficient and simple to implement, for modern demand-driven context-sensitive points-to analyses. We have shown experimentally that tracing CFL-reachability is very effective in avoiding re-analysis of points-to information in Java. Our next step is to study the behaviour of real-world changes and to integrate our analysis into an interactive programming environment. We want to study changes made by real programmers, so that the sequence of changes we test will reflect more accurately modifications likely to be made in practice.

Acknowledgements. This research is supported by Australian Research Grants, DP0987236 and DP130101970.

References

1. M. Acharya and B. Robinson. Practical change impact analysis based on static program slicing for industrial software systems. In *ICSE'11*.

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*.
3. M. G. Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Trans. Program. Lang. Syst.*, 12(3), 1990.
4. M. G. Burke and B. G. Ryder. A critical analysis of incremental iterative data flow analysis algorithms. *IEEE Trans. Software Eng.*, 16(7), 1990.
5. M. D. Carroll and B. G. Ryder. Incremental data flow analysis via dominator and attribute updates. In *POPL'88*.
6. M. Ceccarelli, L. Cerulo, G. Canfora, and M. Di Penta. An eclectic approach for change impact analysis. In *ICSE'10*.
7. S. Chaudhuri. Subcubic algorithms for recursive state machines. In *POPL'08*.
8. R. Goeritzer. Using impact analysis in industry. In *ICSE'11*.
9. B. Hardekopf and C. Lin. Flow-sensitive pointer analysis for millions of lines of code. In *CGO'11*.
10. B. Hardekopf and C. Lin. Semi-sparse flow-sensitive pointer analysis. In *POPL'09*.
11. N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *PLDI'01*.
12. V. Kahlon. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In *PLDI'08*.
13. J. Kodumal and A. Aiken. Banshee: A scalable constraint-based analysis toolkit. In *SAS'05*.
14. J. Kodumal and A. Aiken. The set constraint/CFL reachability connection in practice. In *PLDI'04*.
15. W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *PLDI'92*.
16. S. Lehnert. A taxonomy for software change impact analysis. In *IWPSE-EVOL'11*.
17. O. Lhoták and L. Hendren. Scaling Java points-to analysis using SPARK. In *CC'03*.
18. T. J. Marlowe and B. G. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *POPL'90*.
19. P. H. Nguyen and J. Xue. Interprocedural side-effect analysis and optimisation in the presence of dynamic class loading. In *ACSC'05*.
20. L. L. Pollock and M. L. Soffa. An incremental version of iterative data flow analysis. *IEEE Trans. Software Eng.*, 15(12), 1989.
21. X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of Java programs. In *OOPSLA'04*.
22. T. Reps. Program analysis via graph reachability. In *ILPS'97*.
23. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL'95*.
24. D. Saha and C. Ramakrishnan. Incremental and demand-driven points-to analysis using logic programming. In *PPDP'05*.
25. L. Shang, Y. Lu, and J. Xue. Fast and precise points-to analysis with incremental CFL-reachability summarisation: preliminary experience. In *ASE'12*.
26. L. Shang, X. Xie, and J. Xue. On-demand dynamic summary-based points-to analysis. In *CGO'12*.
27. M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI'06*.
28. M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for Java. In *OOPSLA'05*.
29. Y. Sui, Y. Li, and J. Xue. Query-directed adaptive heap cloning for optimizing compilers. In *CGO'13*.
30. Y. Sui, D. Ye, and J. Xue. Static memory leak detection using full-sparse value-flow analysis. In *ISSTA'12*.
31. Y. Sui, S. Ye, J. Xue, and P.-C. Yew. SPAS: scalable path-sensitive pointer analysis on full-sparse SSA. In *APLAS'11*.
32. R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot: a java bytecode optimization framework. In *CASCON'10*.
33. J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI'04*.
34. R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *PLDI'95*.
35. X. Xiao and C. Zhang. Geometric encoding: forging the high performance context sensitive points-to analysis for Java. In *ISSTA'11*.
36. G. Xu, A. Rountev, and M. Sridharan. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *ECOOP'09*.
37. D. Yan, G. Xu, and A. Rountev. Demand-driven context-sensitive alias analysis for Java. In *ISSTA'11*.
38. M. Yannakakis. Graph-theoretic methods in database theory. In *PODS'90*.
39. 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*.
40. J.-S. Yur, B. G. Ryder, and W. Landi. An incremental flow- and context-sensitive pointer aliasing analysis. In *ICSE'99*.
41. X. Zheng and R. Rugina. Demand-driven alias analysis for C. In *POPL'08*.