Fast and Precise Points-to Analysis for IDEs with CFL-Reachability Summaries

Lei Shang Yi Lu Jingling Xue
University of New South Wales, 2052, Australia
{shangl,ylu,jingling}@cse.unsw.edu.au

ABSTRACT

We introduce a points-to analysis for Java, called EMU, that enables developers to perform pointer-related queries in programs undergoing constant changes in IDEs. EMU achieves its fast response times by adopting a modular approach to incrementally updating method summaries upon code changes: the points-to information in a method is summarised indirectly by CFL reachability rather than directly by points-to sets. Thus, the impact of a code change made in a method is localised, requiring only its affected part to be re-summarised just to reflect the change. EMU achieves its precision by being context-sensitive (for both method invocation and heap abstraction) and field-sensitive. Our evaluation shows that EMU can be promisingly deployed in IDEs where the changes are small.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; F.3.2 [**Logic and Meaning of Programs**]: Program Analysis

General Terms

Algorithms, Languages, Experimentation, Performance

Keywords

Points-to analysis, CFL reachability, Summarisation

1. INTRODUCTION

Points-to analysis is widely used in static analysis tools. It is especially important for object-oriented languages in which the extensive uses of pointer-induced heap accesses, subtyping and dynamic dispatch make it hard to understand value-flow in a program. Studies over several decades have focused on obtaining precise points-to information efficiently, with impressive progress in points-to analysis for Java [6, 7, 8, 9]. However, existing algorithms are not formulated to be used in environments like IDEs where the software is still being developed. In such environments, a points-to analysis must (1) work well in response to small but frequent edits (**Frequent Code Changes**); (2) handle each code change quickly without disrupting developer productivity (**Non-Intrusiveness**), and (3) answer queries as precisely as possible within a small time budget (**Demand Queries with Limited Budgets**).

We introduce a new approach, called EMU, to points-to analysis for Java that simultaneously satisfies these three constraints. There are three contributions. First, by leveraging recent advances on

Allocation sites	o		
Local variables	v		
Global variables	g		
Instance fields	f		
Call sites	i		
Nodes	n	$::=$	$o \mid v \mid g$
Labels	l	$::=$	new | assign | ld(f) | st(f) | entry$_i$ | exit$_i$ | assignglobal
Statements	$stmt$	$::=$	$n \xleftarrow{l} n$
Programs	pro	$::=$	$\varnothing \mid pro \cup \{stmt\}$

Figure 1: An abstraction of Java programs

demand-driven points-to analysis [6, 7, 9], EMU is the first formulated for IDEs by Context-Free-Language (CFL) reachability. Second, EMU achieves efficiency and precision by adopting a modular approach to incrementally updating method summaries. To achieve efficiency, method summaries are expressed indirectly by CFL reachability rather than directly by points-to sets, so that the impact of a code change in a method is bounded in EMU (requiring only its affected part to be re-summarised) rather than unbounded traditionally [2, 10] (requiring program-wise updates in the worst case). To achieve precision, EMU is context-sensitive for (method calls and heap abstraction) and field-sensitive. Finally, we have implemented EMU in Soot, a Java analysis framework integrated into the Eclipse IDE. We describe our evaluation with a representative client, `SafeCast`, using seven Java programs. For small changes, such as adding/deleting statements, EMU can answer each query under 0.054 secs on average and under 0.87 secs in the worst case.

2. CFL-REACHABILITY-BASED ANALYSIS

CFL reachability [6, 7, 9] is an extension of graph reachability [4]. As our analysis is flow-insensitive, a program is represented by a *Pointer Assignment Graph* (PAG), with threes types of nodes, objects, and local/global variables. Every edge is oriented in the direction of value flow, representing a statement as a label in Figure 1.

Array elements are modeled by collapsing all elements into a special field arr. No two classes (methods) contain the same identically named global (local) variable. Figure 2 gives an example, providing an abstraction for the Java container pattern. Figure 3 shows its PAG: (1) o_i denotes the abstract object o created at the allocation site in line i; (2) ret, $tmp1$ and $tmp2$ are temporaries; and (3) $this_m$ denotes the "$this$" parameter of method m.

Each realisable *flowsTo*-path in a PAG has a string formed by concatenating in order the labels of edges in the path, where ld/st 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 *flowsTo*-path from the object to the variable. Two variables x and y may be aliases, denoted $x\ alias\ y$, if an object *flowsTo* both x and y.

Let us see how to discover o_5 as a pointed-to target for $t1$. In Figure 2, o_{22} flows to $h1$, which is the actual parameter passed

```java
1 class Hashtable {
2   Object[] key;
3   Object[] val;
4   Hashtable() {
5     Object[] s1 = new Object[MAXSIZE];
6     Object[] s2 = new Object[MAXSIZE];
7     this.key = s1;
8     this.val = s2;}
9   void put(Object k, Object v) {
10    int index = k.hashCode();
11    Object[] t1 = this.key;
12    t1[index] = k;
13    Object[] t2 = this.val;
14    t2[index] = v;}
15  Object get(Object k) {
16    int index = k.hashCode();
17    Object[] t = this.val;
18    return t[index]; }}
19 class Element{ Object f; }
20 class Main {
21   static void main(...) {
22     Hashtable h1 = new Hashtable();
23     Element e1 = new Element();
24     e1.f = new String("hello");
25     String k1 = "first";
26     h1.put(k1, e1);
27     Hashtable h2 = new Hashtable();
28     Element e2 = new Element();
29     e2.f = new String("world");
30     String k2 = "second";
31     h2.put(k2, e2);
32     Element e3 = (Element) h1.get(k1);
33     Element e4 = (Element) h2.get(k2);
34     Object m = e3.f; }}
```

Figure 2: A Java example

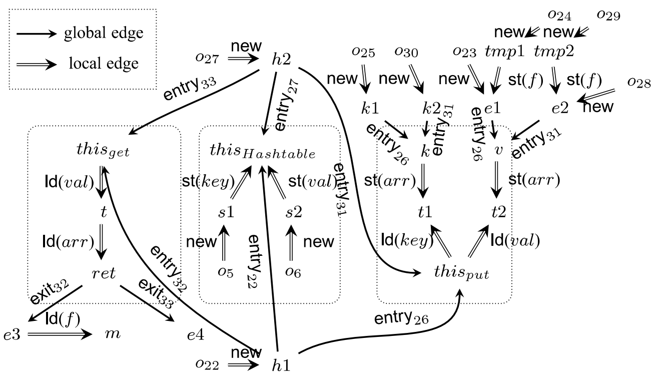


Figure 3: PAG for the Java example.

to the formal parameter $this_{Hashtable}$ of constructor `Hashtable` and $this_{put}$ of `put`. So $this_{Hashtable}$ *alias* $this_{put}$ because o_{22}:

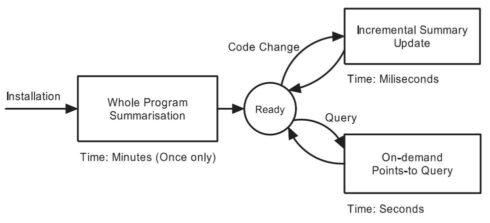
$$this_{Hashtable}\ \overline{\textsf{assign}}\ h1\ \overline{\textsf{new}}\ o_{22}\ \textsf{new}\ h1\ \textsf{assign}\ this_{put}$$

For each edge $x \xleftarrow{\ell} y$, its inverse edge is $y \xleftarrow{\bar{\ell}} x$. We then know that o_5 *flowsTo* $t1$ (so $t1$ points to o_5) due to the *flowsTo*-path:

$$o_5\ \textsf{new}\ s1\ \textsf{st}(key)\ this_{Hashtable}\ alias\ this_{put}\ \textsf{ld}(key)\ t1$$

Note that the *flowsTo* and *alias* relations are recursively defined.

Similarly, we find that o_{24} *flowsTo* m and o_{29} *flowsTo* m. When context sensitivity is considered, only the former *flowsTo* path is realisable. In Figure 2, o_{24} is initially inserted by calling `put` in line 26 into `Hashtable` $h1$, which is created in line 22, and later retrieved by calling `get` in line 32 and saved into m. This *flowsTo* path is realisable because $\textsf{entry}_{26} \rightarrow \overline{\textsf{entry}_{26}} \rightarrow \textsf{entry}_{22} \rightarrow \overline{\textsf{entry}_{22}} \rightarrow \textsf{entry}_{32} \rightarrow \textsf{exit}_{32}$ is. In contrast, the *flowsTo* path corresponding to o_{29} *flowsTo* m is not realisable because $\textsf{entry}_{31} \rightarrow \overline{\textsf{entry}_{31}} \rightarrow \textsf{entry}_{27} \rightarrow \overline{\textsf{entry}_{27}} \rightarrow \textsf{entry}_{33} \rightarrow \textsf{exit}_{32}$ is not.



Installation
Whole Program Summarisation
Time: Minutes (Once only)
Ready
Code Change
Incremental Summary Update
Time: Miliseconds
Query
On-demand Points-to Query
Time: Seconds

Figure 4: Structure of EMU.

Directions	s	$::=$	$S_1 \mid S_2$
Call stacks	c	$::=$	$\varnothing \mid c{:}i$
Field stacks	k	$::=$	$\varnothing \mid d \mid -d \mid k{:}k$
Directed fields	d	$::=$	$f \mid \overline{f}$
Caches	Γ	$::=$	$\varnothing \mid \Gamma \cup \langle v, s\rangle \rightarrow \sigma$
Summaries	σ	$::=$	$\varnothing \mid \sigma \cup \langle n, s, k\rangle$
States	ς	$::=$	$\langle n, s, k, c\rangle$

Figure 5: Abstract domains.

3. THE EMU POINTS-TO ANALYSIS

The EMU framework comprises three phases as shown in Figure 4. The key observation is that, as shown in Figure 3, we can distinguish the edges in a PAG: local edges (new, assign, ld or st) and global edges (assignglobal, entry or exit). For each method parameters and return variables, the summarisation process uses the local reachability analysis to construct and maintain a *CFL-reachability summary*. Such summaries enables fast on-demand points-to queries to be answered, as they can be recomputed quickly and independently in response to code changes.

3.1 Whole-Program Summarisation

As a once-off initialisation, we compute and store local CFL-reachability summaries of all methods. We construct deduction rules (similar to [1]) from a program as specified in Figure 1 with the additional abstract domains in Figure 5. The state s indicates the direction in which the analysis traverses: along a $\overline{\textit{flowsTo}}$ path if $s = S_1$ or a *flowsTo* path if $s = S_2$. A *context (call stack)* c is an ordered sequence of call sites. Similarly, a *field stack* k is an ordered sequence of directed fields. Directed fields are used to distinguish the traversal directions of ld and st edges, where $\overline{f}$ is the inverse of f. A cache Γ maps a variable and a direction of the analysis to its summary. A summary σ is a set of context-insensitive local states. The global states ς are used by points-to queries discussed later.

We specify our local reachability analysis using deductive reachability formulations. The reachability analysis is described by a set of deduction rules in the form of: $\langle n, s, k\rangle \Longrightarrow \langle n', s', k'\rangle$, which follow only local edges and are context-independent. Each local edge in a PAG is translated into one or more deduction rules. For example, given a points-to query for variable v in state S_1 with field stack k, we may conclude that v reaches o with field stack k' if we can derive a reachable path: $\langle v, S_1, k\rangle \Longrightarrow \langle o, s, k'\rangle$.

The summarisation is based on the deduction rules given in Figures 6 and 7. Figure 6 shows the construction of summaries by using the local reachability analysis defined in Figure 7.

During this phase, we compute a *flowsTo* summary in S_2 for each parameter and a $\overline{\textit{flowsTo}}$ summary in S_1 for a return, with the field stack being empty initially. The process exhaustively searches for and includes (1) all reachable local objects (always in state S_1) and (2) all reachable local variables connected to an incoming global edge in S_1 or an outgoing global edge in S_2.

$$\frac{\begin{array}{c}\sigma = \{\langle o, S_1, k\rangle \mid \langle v, s, \varnothing\rangle \Longrightarrow \langle o, S_1, k\rangle\} \cup \\ \{\langle n, S_1, k\rangle \mid \langle v, s, \varnothing\rangle \Longrightarrow \langle n, S_1, k\rangle \ \wedge\ n \xleftarrow{\textsf{exit/entry/assignglobal}} n'\} \cup \\ \{\langle n', S_2, k\rangle \mid \langle v, s, \varnothing\rangle \Longrightarrow \langle n, S_2, k\rangle \ \wedge\ n' \xleftarrow{\textsf{exit/entry/assignglobal}} n\}\end{array}}{\textsc{Sum}(v, s) = \sigma}$$

Figure 6: Summarisation by local reachability analysis.

$$\frac{n \xleftarrow{\text{new}} o}{\langle n, S_1, k\rangle \Longrightarrow \langle o, S_1, k\rangle} \qquad \frac{n' \xleftarrow{\text{assign}} n}{\langle n, S_2, k\rangle \Longrightarrow \langle n', S_2, k\rangle}$$

$$\frac{n \xleftarrow{\text{new}} o}{\langle n, S_1, k\rangle \Longrightarrow \langle n, S_2, k\rangle} \qquad \frac{n' \xleftarrow{\text{st}(f)} n}{\langle n, S_2, k\rangle \Longrightarrow \langle n', S_1, k{:}f\rangle}$$

$$\frac{n \xleftarrow{\text{assign}} n'}{\langle n, S_1, k\rangle \Longrightarrow \langle n', S_1, k\rangle} \qquad \frac{n' \xleftarrow{\text{ld}(f)} n}{\langle n, S_2, k{:}f\rangle \Longrightarrow \langle n', S_2, k\rangle}$$

$$\frac{n \xleftarrow{\text{ld}(f)} n'}{\langle n, S_1, k\rangle \Longrightarrow \langle n', S_1, k{:}\overline{f}\rangle} \qquad \frac{n' \xleftarrow{\text{ld}(f)} n \qquad k \neq k'{:}d}{\langle n, S_2, k\rangle \Longrightarrow \langle n', S_2, k{:}{-f}\rangle}$$

$$\frac{\langle n, s, k\rangle \Longrightarrow \langle n'', s'', k''\rangle \quad \langle n'', s'', k''\rangle \Longrightarrow \langle n', s', k'\rangle}{\langle n, s, k\rangle \Longrightarrow \langle n', s', k'\rangle} \qquad \frac{n \xleftarrow{\text{st}(f)} n'}{\langle n, S_2, k{:}\overline{f}\rangle \Longrightarrow \langle n', S_1, k\rangle}$$

$$\frac{n \xleftarrow{\text{st}(f)} n' \qquad k \neq k'{:}d}{\langle n, S_2, k\rangle \Longrightarrow \langle n', S_1, k{:}{-\overline{f}}\rangle}$$

Figure 7: Local reachability analysis.

$$\text{(local-edge)}\ \frac{stmt = n \xleftarrow{\text{new/assign/ld/st}} n' \qquad \textsc{Update}(\Gamma, n, S_2) = \Gamma' \qquad \textsc{Update}(\Gamma', n', S_1) = \Gamma''}{\textsc{Update}(\Gamma, stmt) = \Gamma''}$$

$$\text{(entry)}\ \frac{stmt = n \xleftarrow{\text{entry}} v \qquad \textsc{Update}(\Gamma, v, S_1) = \Gamma'}{\textsc{Update}(\Gamma, stmt) = \Gamma'}$$

$$\text{(exit)}\ \frac{stmt = v \xleftarrow{\text{exit}} n \qquad \textsc{Update}(\Gamma, v, S_2) = \Gamma'}{\textsc{Update}(\Gamma, stmt) = \Gamma'}$$

$$\text{(global-from)}\ \frac{stmt = g \xleftarrow{\text{assignglobal}} v \qquad \textsc{Update}(\Gamma, v, S_1) = \Gamma'}{\textsc{Update}(\Gamma, stmt) = \Gamma'}$$

$$\text{(global-to)}\ \frac{stmt = v \xleftarrow{\text{assignglobal}} g \qquad \textsc{Update}(\Gamma, v, S_2) = \Gamma'}{\textsc{Update}(\Gamma, stmt) = \Gamma'}$$

$$\text{(update)}\ \frac{\begin{array}{c}\textsc{Sum}(n, s) = \langle n_1, s_1, k_1\rangle, .., \langle n_j, s_j, k_j\rangle \\ \Gamma' = \Gamma[\langle n_1, \overline{s_1}\rangle \rightarrow \textsc{Sum}(n_1, \overline{s_1})]..[\langle n_j, \overline{s_j}\rangle \rightarrow \textsc{Sum}(n_j, \overline{s_j})]\end{array}}{\textsc{Update}(\Gamma, n, s) = \Gamma'}$$

Figure 8: Summary update for code changes.

For `put` in `Hashtable`, our summarisation produces:

$$\begin{aligned}\Gamma(\langle this_{put}, S_2\rangle) &= \langle k, S_1, -key{:}-\overline{arr}\rangle, \langle v, S_1, -val{:}-\overline{arr}\rangle \\ \Gamma(\langle v, S_2\rangle) &= \langle this_{put}, S_1, arr{:}\overline{val}\rangle \\ \Gamma(\langle k, S_2\rangle) &= \langle this_{put}, S_1, arr{:}\overline{key}\rangle\end{aligned}$$

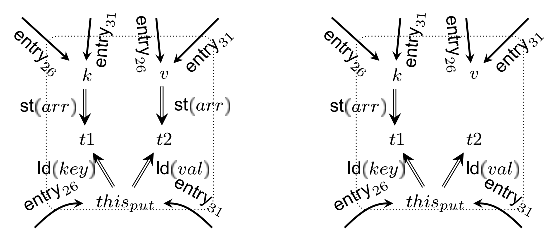
3.2 Incremental Summary Update

After its installation, EMU enters into a ready state, as shown in Figure 4. If a code change is made in a method, then its local reachability summaries are updated. Existing incremental points-to analyses [3, 5, 11] typically cache the points-to sets of the variables queried earlier and update (often approximately) all affected points-to sets after a code change, leading to unbounded (slow) and imprecise propagation of changed points-to information. Since our summaries store local reachability relations, we only need to update the affected summaries for the modified methods. Such modularity leads to a fast and precise analysis suitable for IDEs.

The summaries for all modified methods are updated independently. To update a method, the rules given in Figure 8 are applied. The (update) rule performs the actual re-summarisation while the others determine what summaries are affected and need to be recomputed for different types of edges. We do not distinguish edge insertions and deletions, by treating both as edge changes.

Changes of all local edges are treated uniformly in (local-edge), where we update summaries of variables that n may flow to (in S_2) and summaries of variables that n' may points to (in S_1). Then in (update), we first search for the affected variables in the given direction, then recompute their summaries in the reversed direction (note that $\overline{S_1} = S_2$ and $\overline{S_2} = S_1$), and finally, update the cache.

Changes of global edges are treated individually. In (entry), we recompute the summaries of local variables that v may point to.



k reaches $this_{put}$	k reaches $this_{put}$ (unchanged)
v reaches $this_{put}$	v reaches null (changed)
$this_{put}$ reaches k, v	$this_{put}$ reaches k (changed)
(a) Before	(b) After

Figure 9: Summaries (with field stacks omitted) before and after deleting "`t2[index] = v`" in `put` in line 14 in Figure 2.

$$\text{(local)}\ \frac{\langle n, s\rangle \notin dom(\Gamma) \qquad \langle n, s, k\rangle \Longrightarrow \langle n', s', k'\rangle}{\Gamma, \langle n, s, k, c\rangle \longrightarrow \Gamma, \langle n', s', k', c\rangle}$$

$$\text{(global)}\ \frac{\langle n, s, c\rangle \longrightarrow \langle n', s', c'\rangle}{\Gamma, \langle n, s, k, c\rangle \longrightarrow \Gamma, \langle n', s', k, c'\rangle}$$

$$\text{(summary)}\ \frac{\langle n', s', k'\rangle \in \Gamma(\langle n, s\rangle) \qquad -d \notin k{:}k'}{\Gamma, \langle n, s, k, c\rangle \longrightarrow \Gamma, \langle n', s', k{:}k', c\rangle}$$

$$\text{(transitivity)}\ \frac{\Gamma, \varsigma \longrightarrow \Gamma, \varsigma'' \qquad \Gamma, \varsigma'' \longrightarrow \Gamma, \varsigma'}{\Gamma, \varsigma \longrightarrow \Gamma, \varsigma'}$$

Figure 10: Summary-based points-to analysis.

We do not consider n that is a parameter of another method whose summaries are in the cache and not affected. Similarly, in (exit), we only need to update summaries of local variables that v may flow to, because n is a return variable in another method.

For changes made on a global assignment in (global-from) and (global-to), we consider only the local variables (if any) involved in the assignment. Otherwise, then no summaries are affected.

To re-summarise `put` in Figure 9, we apply (local-edge) to find that the summaries of v and $this_{put}$ are affected but not k:

$$\begin{aligned}\Gamma(\langle this_{put}, S_2\rangle) &= \langle k, S_1, -key{:}-\overline{arr}\rangle \\ \Gamma(\langle v, S_2\rangle) &= \varnothing \\ \Gamma(\langle k, S_2\rangle) &= \langle this_{put}, S_1, arr{:}\overline{key}\rangle\end{aligned}$$

3.3 On-Demand Points-to Queries

With up-to-date summaries, EMU answers on-demand points-to queries by using local summaries and traversing global edges. EMU performs on-demand points-to analysis as described in Section 2 except it uses summaries to speed up the analysis.

Figure 10 gives the deduction rules for our summary-based on-demand points-to analysis in the form of: $\Gamma, \varsigma \longrightarrow \Gamma, \varsigma'$ where Γ contains all up-to-date summaries. We distinguish two types of field stacks: *local field stacks* used in method summaries and *global field stacks* used here for on-demand points-to analysis.

During points-to analysis, a variable v points to an object o in context c if we can derive a points-to path (or a $\overline{\textit{flowsTo}}$ path): $\Gamma, \langle v, S_1, \varnothing, \varnothing\rangle \longrightarrow \Gamma, \langle o, S_1, \varnothing, c\rangle$ where the final global field stack is empty—all fields introduced by ld and st edges must have been matched up. The local field stack that appears in a summary may contain specially annotated fields (i.e., $-d$) to be matched on the global field stack. We may remove a field from the global field stack by defining a syntactical equivalence on two types of field stacks: $k{:}d{:}-d{:}k' \equiv k{:}k'$ $\quad \varnothing{:}k \equiv k$ $\quad k{:}\varnothing \equiv k$ which is used by the (summary) rule to merge both.

Given a points-to query for variable v, we find its point-to set (denoted as $\textsc{Pts}(v)$) by deriving all possible points-to paths:

$$\textsc{Pts}(v) = \{(o, c) \mid \Gamma, \langle v, S_1, \varnothing, \varnothing\rangle \longrightarrow \Gamma, \langle o, S_1, \varnothing, c\rangle\}$$

$$\frac{n \xleftarrow{\text{assignglobal}} n'}{\langle n, S_1, c\rangle \longrightarrow \langle n', S_1, \varnothing\rangle} \qquad \frac{n' \xleftarrow{\text{assignglobal}} n}{\langle n, S_2, c\rangle \longrightarrow \langle n', S_2, \varnothing\rangle}$$

$$\frac{n \xleftarrow{\text{exit}_i} n'}{\langle n, S_1, c\rangle \longrightarrow \langle n', S_1, c{:}i\rangle} \qquad \frac{n' \xleftarrow{\text{exit}_i} n}{\langle n, S_2, c{:}i\rangle \longrightarrow \langle n', S_2, c\rangle}$$

$$\frac{n \xleftarrow{\text{entry}_i} n'}{\langle n, S_1, c{:}i\rangle \longrightarrow \langle n', S_1, c\rangle} \qquad \frac{n' \xleftarrow{\text{exit}_i} n}{\langle n, S_2, \varnothing\rangle \longrightarrow \langle n', S_2, \varnothing\rangle}$$

$$\frac{n \xleftarrow{\text{entry}_i} n'}{\langle n, S_1, \varnothing\rangle \longrightarrow \langle n', S_1, \varnothing\rangle} \qquad \frac{n' \xleftarrow{\text{entry}_i} n}{\langle n, S_2, c\rangle \longrightarrow \langle n', S_2, c{:}i\rangle}$$

Figure 11: Global reachability analysis.

In Figure 10, (local) is used for queried local variables that have no summaries (local variables other than method parameters and returns), in which case, we need to perform a local reachability analysis. The (global) rule deals with context-dependent global edges by a global reachability analysis specified in Figure 11. The (summary) rule looks up and reuses the existing summaries available in the cache and requires that any to-be-matched fields in the local field stack of a summary to be matched on the global field stack.

Figure 11 gives the rules for the global reachability analysis: $\langle n, s, c\rangle \longrightarrow \langle n', s', c'\rangle$ to handle context-dependent global edges.

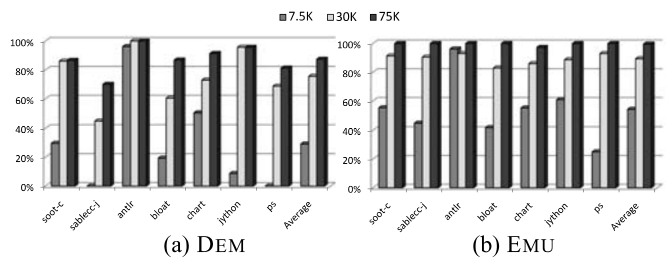
4. EVALUATION

We evaluate the suitability of EMU for deployment in an IDE in terms of its efficiency and precision in response to small code changes. For a program, we randomly selected four kinds of changes (each with 10 statements) corresponding to adding/deleting assignments and calls. We describe our evaluation with a representative client, `SafeCast`, which checks for the safety of downcasts, using seven Java programs from the Dacapo benchmark suite. We exclude the downcasts provably safe by a context-insensitive analysis to show that context sensitivity is critical to achieving higher precision needed. We compare EMU with DEM, a state-of-the-art demand-driven refinement-based points-to analysis [7], both implemented in Soot. All experiments were conducted on an Intel Xeon 3.0GHz processor with 16 GB memory, running RedHat Enterprise Linux 5 (kernel 2.6.18) and Sun JDK 1.6.0.

Benchmark	#M (K)	#Q	Change	DEM (secs) BT	DEM (secs) CT	EMU (secs) IT	EMU (secs) UT	EMU (secs) DT
soot-c	10.4	973	del	43.3	131.5	41.2	0.015	47.4
			add	43.0	125.5		0.013	44.6
			delC	43.0	125.4		0.010	44.8
			addC	44.7	135.7		0.011	48.7
sablecc-j	21.4	358	del	127.3	76.6	63.6	0.016	31.8
			add	126.7	77.0		0.013	32.0
			delC	126.9	71.7		0.012	31.5
			addC	127.9	79.3		0.013	31.7
antlr	12.9	281	del	35.9	15.4	12.5	0.015	4.8
			add	35.9	15.5		0.014	4.8
			delC	36.6	15.1		0.020	4.8
			addC	36.0	15.4		0.013	4.8
bloat	10.8	1221	del	43.8	143.9	24.2	0.009	57.7
			add	43.2	145.8		0.015	57.7
			delC	43.7	144.9		0.011	57.9
			addC	46.0	148.1		0.012	58.9
chart	17.4	682	del	143.0	191.8	74.5	0.011	68.9
			add	140.1	192.4		0.016	69.0
			delC	141.3	191.9		0.007	68.9
			addC	142.3	194.1		0.005	69.6
jython	27.5	524	del	38.3	27.5	28.0	0.015	5.2
			add	38.6	27.5		0.016	5.2
			delC	38.8	26.9		0.015	5.1
			addC	38.7	27.6		0.014	5.2
ps	13.5	660	del	128.4	96.1	63.4	0.011	42.3
			add	127.7	97.5		0.011	44.3
			delC	124.8	104.7		0.007	46.8
			addC	129.7	94.7		0.007	42.3

Table 1: Analysis times of `SafeCast` by DEM and EMU.

Table 1 gives the analysis times from DEM and EMU. The "#M (K)" column gives the number of reachable methods for each program. The "#Q" column gives the number of queries. In the "Change" column, the four tests are performed. The analysis times are given in the last five columns. Let us consider DEM first. After a code change, the time taken per query is given by BT + CT/#Q, where BT represents the time spent on rebuilding the PAG and CT the time spent to answer all queries. The cost for rebuilding the PAG after each code change is too high to to be directly deployed in IDEs. As for EMU, Whole-Program Summarisation happens only once for a program. IT gives the time for computing the local CFL-reachability summaries for all methods in its PAG. After each code change, the (amortised) time per query is given by DT/#Q if incremental summary update can complete before on-demand query starts and is bounded by (UT+DT)/#Q otherwise. Here, UT gives the time for updating the summaries affected by the code change and DT gives the time in answering the queries issued. In addition, EMU is about 3× faster in answering queries than DEM. EMU can answer each query under 0.054 secs on average and under 0.87 secs in the worst case. There are no significant performance variations among the four tests due to the modularity of our approach.



(a) DEM (b) EMU

Figure 12: Precision of `SafeCast`.

We measure the precision with different budgets against an exhaustive analysis. Figure 12 compares DEM and EMU with the precision of `SafeCast` being defined as the percentage of queries that gives the same answers as the exhaustive analysis under three budgets, 7.5K, 30K and 75K. For DEM, the average precision percentages are 29.16%, 75.55% and 87.38%. respectively. For EMU, better ones, 54.19%, 89.34% and 99.61%, are achieved, respectively.

We measured the memory usage of the whole JVM heap with and without summaries. EMU consumes slightly more memory than DEM, ranging from 21MB to 70MB across the seven benchmarks with an average of 46.25MB. This translates into percentage increases ranging 3.27% to 16.24% with an average of 6.94%.

5. ACKNOWLEDGEMENTS

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

6. REFERENCES

[1] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *PLDI'01*.

[2] V. Kahlon. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In *PLDI'08*.

[3] J. Kodumal and A. Aiken. Banshee: A scalable constraint-based analysis toolkit. In *SAS'05*.

[4] T. Reps. Program analysis via graph reachability. In *ILPS'97*.

[5] D. Saha and C. Ramakrishnan. Incremental and demand-driven points-to analysis using logic programming. In *PPDP'05*.

[6] L. Shang, X. Xie, and J. Xue. On-demand dynamic summary-based points-to analysis. In *CGO'12*.

[7] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI'06*.

[8] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI'04*.

[9] D. Yan, G. Xu, and A. Rountev. Demand-driven context-sensitive alias analysis for Java. In *ISSTA'11*.

[10] 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*.

[11] J.-S. Yur, B. G. Ryder, and W. Landi. An incremental flow- and context-sensitive pointer aliasing analysis. In *ICSE'99*.