Probabilistic Points-to Analysis for Java

Qiang Sun[1], Jianjun Zhao[1,2], and Yuting Chen[2]

[1] Department of Computer Science and Engineering
[2] School of Software
Shanghai Jiao Tong University
800 Dongchuan Road, Shanghai 200240, China

Abstract. Probabilistic points-to analysis is an analysis technique for defining the probabilities on the points-to relations in programs. It provides the compiler with some optimization chances such as speculative dead store elimination, speculative redundancy elimination, and speculative code scheduling. Although several static probabilistic points-to analysis techniques have been developed for C language, they cannot be applied directly to Java because they do not handle the classes, objects, inheritances and invocations of virtual methods. In this paper, we propose a context-insensitive and flow-sensitive *probabilistic points-to analysis for Java* (JPPA) for statically predicting the probability of points-to relations at all program points (i.e., points before or after statements) of a Java program. JPPA first constructs an interprocedural control flow graph (ICFG) for a Java program, whose edges are labeled with the probabilities calculated by an algorithm based on a static branch prediction approach, and then calculates the probabilistic points-to relations of the program based upon the ICFG. We have also developed a tool called *Lukewarm* to support JPPA and conducted an experiment to compare JPPA with a traditional context-insensitive and flow-sensitive points-to analysis approach. The experimental results show that JPPA is a precise and effective probabilistic points-to analysis technique for Java.

Keywords: points-to analysis, probability, Java.

1 Introduction

Points-to analysis is an analysis technique which is widely used in compiler optimization and software engineering [1,2]. The goal of points-to analysis is to compute points-to relations between variables of pointer types and their allocation sites. Context-sensitivity and flow-sensitivity are two major aspects of points-to analysis for improving the precision of the analysis [3]. While the context-sensitive points-to analysis [4,5] distinguishes the different contexts in which a method is invoked and then analyzes the method individually for each context, the flow-sensitive points-to analysis [6,7] takes into account the control flows inside a program or a method, and computes the solutions (i.e., points-to relations) for the program points on the control flow of each method. Especially, a flow-sensitive points-to analysis helps deduce that for each points-to relation whether it *definitely* exists or *maybe* exists at any program point.

Probabilistic points-to analysis [8], which defines the probability of each points-to relation, provides the compiler with some optimization chances. With the probabilistic points-to relations, a compiler may perform speculative dead store elimination, redundancy elimination, and code scheduling [9,10,11]. For example, by speculation a compiler with a recovery mechanism may characterize a variable as redundant if its points-to relation is of high probability of the same as those of other variables. A probabilistic points-to analysis usually follows two steps to compute, during the executions of a program, the quantitative information of the likelihood a points-to relation *maybe* holds. First, all the execution paths and their frequencies are collected at runtime. Second, the points-to relations are deduced and the probabilities of points-to relations are computed according to the path frequencies. However, a challenge about decreasing the costs of computation of the probabilities in a large-scale program still exists because a program may run hundreds of times and a large amount of time and memory can be consumed.

One possible solution to above problem is to conduct static probabilistic points-to analyses of programs. Although several static probabilistic points-to analysis techniques have been developed for C language, they cannot be applied directly to the analysis of Java programs due to the differences between Java and C languages. For example, the techniques for analyzing C function pointers usually compute the points-to set for each function pointer and the probabilities of the elements inside. We could not use these techniques to analyze the invocation of Java virtual methods, while a Java virtual method is invoked by an object, and an analysis of the invocation of the method requires the determination of the receiver and the exploration of the distributions over the points-to set for the receiver.

In this paper, we propose a context-insensitive and flow-sensitive *probabilistic points-to analysis for Java* (JPPA) for statically predicting the probability of each points-to relation at each program point of a Java program. JPPA first constructs a call graph through the traditional static analysis [12,13,14], and then builds an intraprocedural control flow graph (CFG) with probabilities for each method in the call graph. The probabilities can be computed according to the result of a static branch prediction. After that JPPA combines the call graph and CFGs to construct an interprocedural CFG (ICFG), and then carries out the data-flow analysis on the ICFG for constructing the probabilistic points-to graph at each program point. In JPPA, a points-to relation is not confined to yes/no but is associated with a real number representing the probability. We have also implemented JPPA with a tool called *Lukewarm* and conducted an experiment to evaluate the effectiveness of JPPA. The experimental results show that, for our benchmark programs, JPPA provides a cost-effective manner in computing the probabilistic points-to relations in Java programs.

This paper makes the following contributions:

- **Abstraction.** We define the probabilities on the edges of ICFG, and these probabilities, which share one destination node, forms a distribution. We also use a discrete probability distribution to represent a points-to set with probabilities.
- **Analysis Approach.** We develop a probabilistic points-to analysis technique for Java named JPPA, which takes into account the object-oriented features such as inheritance and polymorphism. Especially, in order to improve the performance of analysis, JPPA computes a partial probabilistic points-to graph before parameter

```java
1  public class Shape {
2      public Double area = null;
3      public Double set(Double s) {
4          this.area = s;
5          return s;
6      }
7      public static void main(String args[]) {
8          int a = randomInt();
9          int b = randomInt();
10         Shape p = null;
11         if(a>0 && b>0) {
12             p = new Circle();           //o1
13         }
14         else {
15             p = new Square();           //o2
16         }
17         Double q =
               new Double(Math.abs(a*b)); //o3
18         Double r = p.set(q);
19         System.out.println(p.area);
20     }
21     private static double randomInt() {
22         return Math.floor(Math.
                   random()*11-5);
23     }
24 }
```

```java
25 public class Circle extends Shape {
26     public Double area = null;
27     public Double radius = null;
28     public Double set(Double s) {
29         super.area = new Double(0);      //o4
30         this.area = s;
31         this.radius =
             new Double(Math.sqrt(s/3.14)); //o5
32         return this.radius;
33     }
34 }
```

```java
35 public class Square extends Shape {
36     public Double sideLength = null;
37     public Double set(Double s) {
38         this.area = s;
39         this.sideLength =
             new Double(Math.sqrt(s));      //o6
40         return this.sideLength;
41     }
42 }
```

Fig. 1. A sample Java program

passing; in order to improve the precision of analysis, JPPA computes the probabilities over the edges of ICFG dynamically. We have also developed a tool to support the JPPA approach.

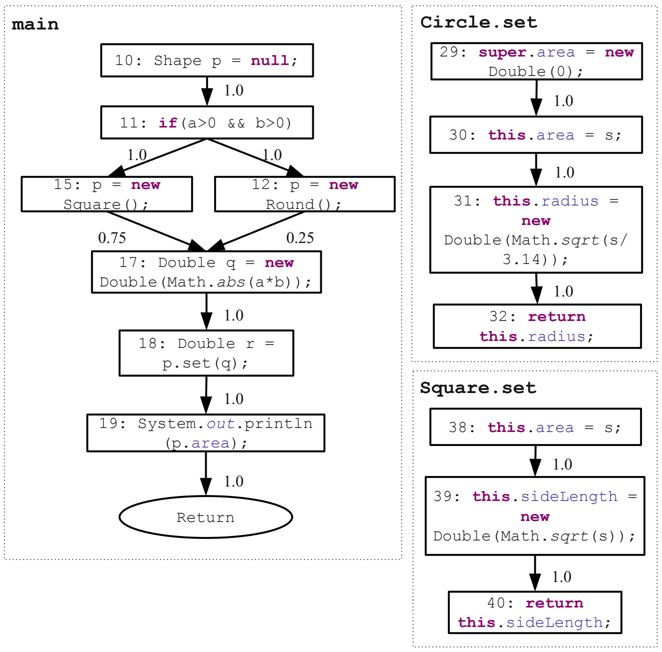
– **Analysis Results.** We have conducted an experiment to evaluate JPPA, and the experimental results show that our analysis can be used to compute precisely the probabilistic points-to information in Java programs.

2 Example

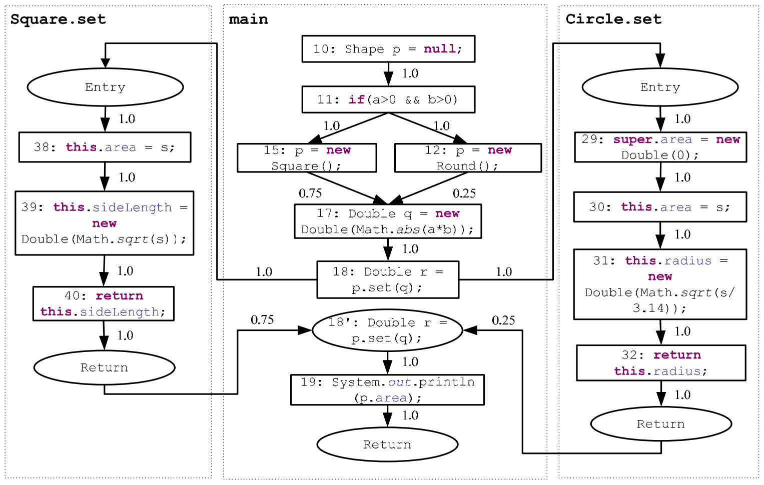
We next present an example to illustrate how our static probabilistic points-to analysis works. Fig. 1 shows a Java program with three classes: `Shape`, `Circle`, and `Square`. Class `Shape` declares shapes and their sizes, and classes `Circle` and `Square` extend `Shape` with two fields `radius` and `sideLength` respectively. The method `randomRealNum()` in class `Shape` randomly generates values to simulate the inputs from the real world. The `main()` method in class `Shape` receives an input and then creates a shape (either a circle or a square) and prints its field of `area`.

Suppose `randomRealNum()` generates the real numbers complying to the uniform distribution in the range $[-5, +5]$. The probabilities of the **if-else** branches (see lines 11-16) in the `main()` method can be easily referred: the `if` branch is of a probability of 0.25 and the `else` branch a probability of 0.75. Suppose the objects created in the sample program in Fig. 1 are `oi (i=1..6).`

Our JPPA analysis consists of four steps. At the beginning, a call graph is constructed through the traditional static analysis [12,13,14] in order to remove the unreachable methods to reduce the analyzing costs. Although the call graph built by static analysis may not be precise enough as it still contains some unreachable methods, JPPA can refine it based on the points-to information. The points-to set $\{o_1, o_2\}$ of variable p at line 18 can be computed through analyzing the statements at lines 12 and 15.



(a) The CFGs for methods main, Circle.set and Square.set



(b) The ICFG for the whole sample program

Fig. 2. The CFGs and ICFG of the sample program in Fig. 1

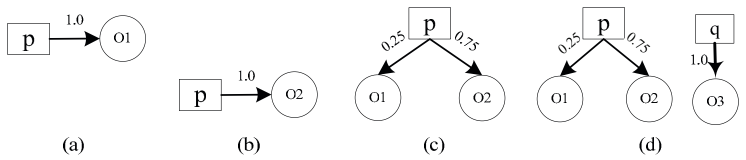


Fig. 3. The probabilistic points-to analysis for method `main` (part I)

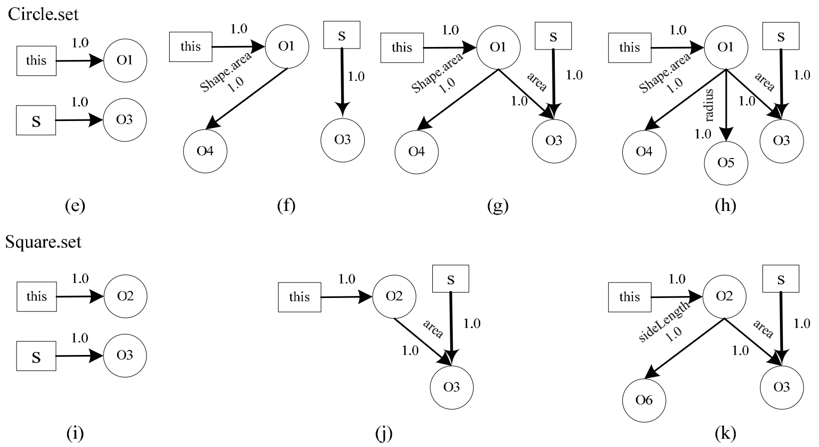
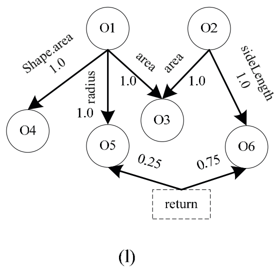


Fig. 4. The probabilistic points-to analysis for methods `Circle.set` and `Square.set`

Through identifying the classes of o_1 and o_2, both the methods `Circle.set()` and `Square.set()` can be invoked at line 18. Therefore, the analyzed methods includes `main()`, `Circle.set()`, and `Square.set()`.

The second step is to construct a CFG for each method in the call graph. Fig. 2 (a) shows the CFGs for methods `main()`, `Circle.set()` and `Square.set()`, in which each node in the CFG represents a statement and each edge is labeled with a real number for the predecessor-dependent probability (see Section 3). Since the traverse of node 15 must be preceded by the traverse of node 11, the probability of the edge from node 11 to node 15 is 1.0; a traverse of node 17 may succeed the traverse of node either 12 or 15 and the probabilities of the **if-else** branches are 0.75 and 0.25, and therefore the probabilities of the edges from nodes 15 and 12 to 17 are 0.75 and 0.25, respectively.

The third step is to combine the CFGs of the methods of interests into an interprocedural CFG (ICFG) according to the call graph. Fig. 2 (b) shows the ICFG of the sample program with the probabilities on the edges. Note that we add some new nodes to the ICFG in order to simplify our discussion. Node 18' copies the node 18 in order to receive the return values of the method invocation `Circle.set()` or `Square.set()`. For each method, an entry node and a return node are added to assign the parameters and return values, respectively. The probabilities on the interprocedural return edges are initialized by 0.0 at first and then adjusted in the final step.



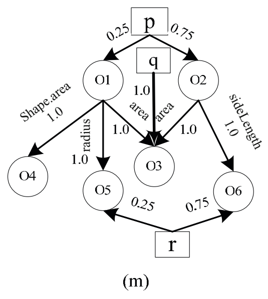


Fig. 5. The probabilistic points-to analysis for method `main` (part II)

The final step is to analyze each node in the ICFG to compute the probabilistic points-to graph. Fig. 3 illustrates the probabilistic points-to analysis of the statements between lines 11 and 17. The probabilistic points-to graphs (Fig. 3 (a) and Fig. 3 (b)) are calculated after analyzing the statements at lines 12 and 15, respectively. Before line 17, Fig. 3 (c) is calculated by merging the two graphs in Fig. 3 (a) and 3 (b). After analyzing the statement at line 17, the graph in Fig. 3 (d) is generated. Because of the analysis in the first step, either `Circle.set()` or `Square.set()` can be invoked in line 18. The probabilities on the interprocedural edges are adjusted according to the points-to probabilities of the receiver objects. The variable `p` points to the objects o_1 and o_2 with the probabilities 0.25 and 0.75 respectively. The probabilities of the edges from the return nodes of methods `Circle.set()` and `Square.set()` to the node 18' are adjusted by 0.25 and 0.75. A detailed algorithm about the calculation of the probabilistic points-to graphs is given in Section 3.

Since the field `area` is overloaded in the class `Circle`, the field declared in class `Shape` is marked as `Shape.area`. The problem does not occur in class `Square` because it inherits `area` from class `Shape`. Fig. 4 manifests the analysis of `Circle.set()` and `Square.set()`, and Fig. 5 shows the analysis from the end of method call to the end of method `main()`. In Fig. 5, the graph (l) is built by merging the two graphs (h) and (k), in which the local variables are eliminated and the return value is marked by the dish square node. The graph (m) represents the points-to graph after the method call that is generated by updating the graph (d) before the method call uses the graph (l) and replacing the return node by `r` node.

3 Probabilistic Points-to Analysis for Java

3.1 Probabilistic Points-to Graph

Traditionally, in order to conduct a points-to analysis of a Java program, three sets are defined [4]:

– Ref: a set containing all reference variables in the program and all static fields in its classes.

– Obj: a set containing the names of all objects created at the object allocation sites.
– $Field$: a set containing instance fields but not the static fields in the classes.

In addition, we define $R \subseteq Ref \cup Obj \times Field$ containing all object references if they belong to Ref or have a form of $\langle o, f\rangle \in Obj \times Field$. Two relations are defined on the three sets: $(r, o) \in Ref \times Obj$ representing that a reference variable r points to an object o and $(\langle o, f\rangle, o') \in (Obj \times Field) \times Obj$ representing that a field f of the object o points to an object o'.

The goal of probabilistic points-to analysis is to compute the probability of each points-to relation holding at every program point. A *program point* is a code location before or after a statement executed. In order to do this, we define an expected probability as follows:

$$Prob(l, d) = \begin{cases} \frac{Expected(l,d)}{Expected(l)} & Expected(l) \neq 0, \\ 0 & Expected(l) = 0. \end{cases}$$

where *d* is a points-to relationship, *Expected(l)* is the times which a program point *l* is expected to turn up on the program execution paths and *Expected(l,d)* is the times which *d* is expected to hold dynamically at a program point *l* [15].

For a given program point *l*, the points-to set of a reference r or $\langle o, f\rangle$ with probabilities satisfies a discrete probability distribution over the object set Obj, and the probabilistic points-to relationships at *l* is a set of the distributions. A discrete probability distribution over a set Obj is a mapping

$$\Delta : Obj \mapsto [0, 1], \sum_{o \in Obj} \Delta(o) = 1.$$

The *support* of Δ is $\lceil \Delta \rceil := \{o \in Obj \mid \Delta(o) > 0\}$. If a points-to relation d is in the form of (r, o) or $(\langle o_1, f\rangle, o_2)$ at the program point *l*, the distributions of the references satisfy $\Delta_r(o) = Prob(l, (r, o))$ or $\Delta_{o_1.f}(o_2) = Prob(l, (\langle o_1, f\rangle, o_2))$.
$\bar{o}$ is the point distribution over Obj satisfying

$$\bar{o}(t) = \begin{cases} 1 & t = o,\ t \in Obj, \\ 0 & otherwise. \end{cases}$$

If Δ_i is a distribution for each i in some finite index set I and $\sum\limits_{i \in I} p_i = 1$, $\sum\limits_{i \in I} p_i \cdot \Delta_i$ is also a distribution $(\sum\limits_{i \in I} p_i \cdot \Delta_i)(o) = \sum\limits_{i \in I} p_i \cdot \Delta_i(o), o \in Obj$. Every distribution can be represented as a linear combination of the point distributions with the form of $\Delta = \sum\limits_{o \in \lceil \Delta \rceil} \Delta(o) \cdot \bar{o}$.

► **Example.** The distribution of variable p before line 18 is represented by $\Delta_p = 0.25\bar{o_1} + 0.75\bar{o_2}$. ◄

In our research, we use *probabilistic points-to graph* (PPG) for probabilistic points-to analysis of Java programs. A PPG is a directed multi-graph whose nodes are the elements belonging to Ref and Obj. Each edge has a probability and represents a probabilistic points-to relation either from a variable to an object or from an object to another object. Specially, an edge from an object o_1 to another o_2 (e.g., the edge from node o2 to node o3 in Fig. 4(j)) means a field of o_1 points to o_2, and holds the field information. Thus a PPG contains a distribution set of the reference variables.

▶ **Example.** For the program point after line 18 in Fig. 1, the PPG (see Fig. 5(m)) shows the distributions of p, q, r and all field references.

$$\Delta_p = 0.25\bar{o_1} + 0.75\bar{o_2} \quad \Delta_r = 0.25\bar{o_5} + 0.75\bar{o_6} \quad \Delta_q = 1.0\bar{o_3}$$
$$\Delta_{o1.radius} = 1.0\bar{o_5} \quad \Delta_{o1.Shape.area} = 1.0\bar{o_4} \quad \Delta_{o1.area} = 1.0\bar{o_3}$$
$$\Delta_{o2.area} = 1.0\bar{o_3} \quad \Delta_{o2.sideLength} = 1.0\bar{o_6} \blacktriangleleft$$

In the research we provide a program with predecessor-dependent probabilities information so that PPGs can be precisely calculated. In order to perform probabilistic points-to analysis, a program is represented by a ICFG (e.g., Fig. 2 (b)) whose edges are labeled with probabilities. We call this probability predecessor-dependent probability to distinguish the branch probability. A branch probability [16] ($edge\_prob(s_i \rightarrow s_j)$) is an estimate of the likelihood that a branch will be taken. For any two statement s_1 and s_2 in a method, the predecessor-dependent probability is an estimate of the likelihood that s_1 directly reaches s_2. The branch probabilities calculated by the branch prediction algorithm [16] can not be directly applied to the merge operation in our framework. The branch probabilities on the edges of the intra-method CFG can be used to compute the predecessor-dependent probabilities. Given a path $(s_0, ..., s_n)$, the path probability is computed by

$$path\_prob(s_0, ..., s_n) = \prod_{i=0}^{n} edge\_prob(s_i \rightarrow s_{i+1})$$

where s_0 represents the method entry statement. The predecessor-dependent probability for the edge (s_i, s_j) is computed by

$$EP((s_i, s_j)) = \frac{\sum path\_prob(s_0, ..., s_i)}{\sum path\_prob(s_0, ..., s_i, s_j)}$$

▶ **Example.** After the branch prediction analysis, we have
$edge\_prob(10 \rightarrow 11) = 1.0, edge\_prob(15 \rightarrow 17) = 1.0, edge\_prob(12 \rightarrow 17) = 1.0,$ $edge\_prob(11 \rightarrow 12) = 0.25, edge\_prob(11 \rightarrow 15) = 0.75.$
The predecessor-dependent probability of the edge from 15 to 17 is calculated as follows:

$$\begin{aligned} EP((15,17)) &= \frac{path\_prob(10,11,15,17)}{path\_prob(10,11,15,17) + path\_prob(10,11,12,17)} \\ &= \frac{1.0 \times 0.75 \times 1.0}{1.0 \times 0.75 \times 1.0 + 1.0 \times 0.25 \times 1.0} = 0.75 \blacktriangleleft \end{aligned}$$

Since in Java there are four ways to assign a value to a reference variable that may change a points-to relation. The statements in these four forms should be analyzed. These forms are:

– Create an object: $v = new\ C$;
– Assign a value: $v = r$;

– Read an instance field : $v = r.f$;
– Write an instance field : $v.f = r$.

▶ **Example.** Nodes 32 and 21 in Fig. 2 (b) create an object and write an instance field, respectively. ◀

3.2 Intraprocedural Analysis

A points-to analysis can be formulated as a data flow framework which includes transfer functions formulating the effect of statements on points-to relations [6]. As a result, our probabilistic points-to analysis framework can be represented by a tuple

$$(L, \sqcup, Fun, P, Q, E, \iota, M, EP)$$

where:

– L is a lattice and a PPG can be regarded as an element of L
– $\sqcup$ is the meet operator
– $Fun \subseteq L \mapsto L$ is a set of monotonic functions
– P is the set of the statements
– $Q \subseteq P \times P$ is the set of flows between statements
– E is the initial set of statements
– ι specifies the initial analysis information
– $M : P \mapsto Fun$ is a map from statements to transfer functions
– $EP : Q \mapsto [0, 1]$ is an predecessor-dependent probability function

The partial order over L is determined by

$$\forall G_1, G_2 \in L, G_1 \sqsubseteq G_2 \ \texttt{iff} \ \forall r \in R, \lceil \Delta_r^{G_1} \rceil \subseteq \lceil \Delta_r^{G_2} \rceil.$$

And the meet operation of two graphs G_1 and G_2 is

$$G_1 \sqcup G_2 = \{p \cdot \Delta_r^{G_1} + (1 - p) \cdot \Delta_r^{G_2} \mid r \in R, p \in [0, 1]\}.$$

Let $f_s \in Fun$ be the transfer function of the statement s, and $G_{in}(s)$ and $G_{out}(s)$ represent the PPGs at the program points before and after the statement s, respectively, we have

$$G_{in}(s) = \begin{cases} \iota & if\ s \in E \\ \bigsqcup\{G_{out}(s') \mid (s', s) \in Q\} & otherwise \end{cases}$$
$$G_{out}(s) = f_s(G_{in}(s))$$

The statement s is associated with the transfer function that transforms $G_{in}(s)$ to $G_{out}(s)$, and the analysis iteratively computes the $G_{in}(s)$ and $G_{out}(s)$ for all nodes until convergence.

In a probabilistic points-to analysis, E only contains the first statement during the program execution and ι is a special probabilistic points-to graph in which all the reference variables point to undefined target (i.e., UND) with total probabilities. Next we describe the transfer functions for assignments and branches.

Table 1. Computing distributions

Statement	Updating the distributions of $G_{out}(s)$
$v = new\ C$	$\varDelta_v^{G_{out}(s)} \leftarrow \bar{o}$
$v = r$	$\varDelta_v^{G_{out}(s)} \leftarrow \varDelta_r^{G_{in}(s)}$
$v = r.f$	$\varDelta_v^{G_{out}(s)} \leftarrow \sum_{o \in \lceil \varDelta_r^{G_{in}(s)} \rceil} \varDelta_r^{G_{in}(s)}(o) \cdot \varDelta_{o.f}^{G_{in}(s)}$
$v.f = r$	$\varDelta_{o.f}^{G_{out}(s)} \leftarrow \varDelta_v^{G_{in}(s)}(o) \cdot \varDelta_r^{G_{in}(s)} + (1 - \varDelta_v^{G_{in}(s)}(o)) \cdot \varDelta_{o.f}^{G_{in}(s)}, o \in \lceil \varDelta_v^{G_{in}(s)} \rceil$

Assignments. For any assignment statement s, there is a corresponding transfer function F_s. F_s takes $G_{in}(s)$ as input and computes the result $G_{out}(s)$. A transfer function first copies $G_{in}(s)$ to $G_{out}(s)$ and then updates $G_{out}(s)$. Table 1 describes transfer functions for assignment statements: the transfer function for $v = new\ C$ updates the distribution $\varDelta_v^{G_{out}(s)}$ with the point distribution of o created by $new\ C$ expression; the transfer function for $v = r$ replaces the distribution $\varDelta_v^{G_{out}(s)}$ with the distribution $\varDelta_r^{G_{in}(s)}$; the transfer function for $v = r.f$ composes the distributions of the field f of all the objects in the *support* set of $\varDelta_r^{G_{in}(s)}$; and the transfer function for $v.f = r$ updates multiple distributions in the form of $o.f$.

A probabilistic points-to analysis also needs to take arrays into account, each of which may contain multiple references. Since an array $array$, is initialized as

$$C\,[\,]\,array \;=\; new\,C\,[n];$$

where n can be either a constant or a variable, it is difficult to infer the range of n. We can obtain the array elements of multiple references and estimate the total number of references they point to if we cannot determine the points-to set of each element in a static manner. When an element o is stored to $array$ (i.e., $array[i] = o$), o is added to the points-to set of $array$, say $Pt(array)$. Then each points-to probability is recalculated

$$p = \frac{1}{|Pt(array)|}$$

where $|Pt(array)|$ is the number of the elements in $Pt(array)$. When $array[i]$ is accessed (e.g., $v = array[i]$), the distribution of v is updated by the distribution of $array$.

Branch. When multiple nodes directly reach a destination node s in a CFG, the *meet* operation is adopted in the calculation of $G_{in}(s)$. Suppose s is the successor of the nodes $s_i (i \in I)$ in a CFG. The following condition is satisfied

$$\sum_{i \in I} EP((s_i, s)) = 1$$

where $EP((s_i, s))$ represents the probability of edge (s_i, s). The *meet* operation can be described by

$$G_{in}(s) = \bigsqcup \{G_{out}(s_i) \mid i \in I\} \triangleq \sum_{i \in I} EP((s_i, s)) \cdot G_{out}(s_i).$$

► **Example.** In an ***if-then-else*** statement, suppose $G_{out}(s_{then})$ and $G_{out}(s_{else})$ are the PPGs at the exit points of the ***then*** and ***else*** branches respectively, p_t and p_f are the probabilities of the ***then*** and ***else*** branches respectively, and $p_t + p_f = 1$. A PPG at s_{join}, the statement succeeding the ***if-then-else*** statement, can be computed by using *meet* operation: $G_{in}(s_{join}) = p_t \cdot G_{out}(s_{then}) + p_f \cdot G_{out}(s_{else})$. ◄

Loop. The loop body B can be unfolded for arbitrary times. The computation can be formulated as

$$G_{in} = \sum_{\alpha \le i \le \beta} p_i \cdot F^i(G_0) + p_0 \cdot G_0, \sum_{\alpha \le i \le \beta} p_i + p_0 = 1$$

where α and β are the upper-bound and lower-bound of iteration number, G_0 represents the initial PPG before entering the loop, F is the transfer function of loop body B, p_0 represents the probability of not entering the loop, p_i represents the probability of i times iteration. Two problems arise when a loop is analyzed: (1) how to estimate the upper- and lower-bounds of iteration number, and (2) how to estimate the probability of some iteration number.

In JPPA, the loop body is unfolded for the upper-bound times. The lower-bound of iteration number is 0. The upper-bound of iteration number is the minimum N, which satisfies $\forall r \in R$, $\lceil \Delta_r^{F^N(G_0)} \rceil = \lceil \Delta_r^{F^{N+1}(G_0)} \rceil$. The probability of each iteration number is $1/(N+1)$.

Exception Handling. The exception handling in Java encapsulates exception in class, uses the exception handling mechanism of ***try-catch-finally*** and gets more robust exception handling code finally. In JPPA framework, the probabilistic points-to analysis can easily go deep into the exception blocks through building the CFG for them. In the ***try-catch-finally*** structure, the exceptions are thrown out from every program point in the try block, then the edges from the program points to the entry point of catch block are generated. In our study we adopt Soot to construct CFGs for ***try-catch-finally*** structures. However, a precise calculation of points-to relation information in exceptions requires obtain all program points that may throw these exceptions, which remains in our future work.

3.3 Interprocedural Analysis

Interprocedural probabilistic points-to analysis analyzes points-to relations crossing the boundary between methods. At each call site, points-to relations are mapped from the actual parameters to the formal parameters, and the results are mapped back to the variables in the caller.

Since a Java program may rely heavily on libraries with a number of irrelevant methods, JPPA adopts RTA algorithm [13] to produce an approximation of the corresponding call graph so that some irrelevant methods can be ignored. After that, JPPA uses the call graph to construct the ICFG. At each call site, two extra nodes are generated for each callee: an entry node recording the passing of parameters, and a return node recording the returns of the callee. Specially, all values returned by the callee are assigned to a unique variable in the return node. JPPA then refines the ICFG by adjusting the probabilities on the interprocedural edges according to the PPGs at all method call sites.

When a method call is analyzed, a partial PPG is propagated through an interprocedural edge to the callee method. JPPA takes a set of actual parameters and a PPG as its inputs, and then computes all the objects that can be accessed by these actual parameters. The partial PPG as a result includes all the points-to relations with the probabilities that may be manipulated by the callee method.

Since JPPA is a context-insensitive analysis without distinguishing the contexts under which a method is invoked, it may share one callee method with different calling contexts. Thus a meet operation can be defined as

$$G_{in}(s_m) = \sum_{cs_i \in CS} EP(e_{cs_i}) \cdot G_{out}(s_{cs_i}), \;\; e_{cs_i} = (s_{cs_i}, s_m)$$

where m is the callee method, $G_{in}(s_m)$ is the PPG before the program point entering the method m, CS denotes the set of call sites at each of which m is invoked, $EP(e_{cs_i})$ denotes the probability of the invocation of m at cs_i, and $G_{out}(s_{cs_i})$ represents the PPG after the passing of parameters at cs_i.

When the return value is assigned to the variable in the caller method, all the points-to relations updated by the callee method need to be reflected upon the PPG after the method invocation.

Virtual Invocation. In Java, virtual method is a method whose behavior can be overridden within an inheriting class by a method with the same signature. The compiler and loader can guarantee the correct correspondence between objects and the methods applied to them.

A virtual method m is usually invoked explicitly or implicitly by a *this* object, and thus $this$ needs to be mapped to the receiver object invoking m. Suppose m is declared in class C and at each call site $cs_i \in CS$, m may be invoked in a form $r_i.m()$ where r_i is the receiver object invoking m at cs_i. At each call site cs_i, there exits an object set

$$set_i = \{o \mid o \in \lceil \Delta_{r_i} \rceil \wedge o.Class \in MatchedClass(C, m)\}$$

where $MatchedClass(C, m)$ is a set of classes containing class C and all its subclasses in each of which m is inherited. The distribution of $this$ can be computed

$$\Delta_{this} = \sum_{cs_i \in CS} \sum_{o \in set_i} \frac{EP(e_{cs_i}) \cdot \Delta_{r_i}(o)}{a} \bar{o},$$
$$a = \sum_{cs_i \in CS} \sum_{o \in set_i} EP(e_{cs_i}) \cdot \Delta_{r_i}(o), \;\; e_{cs_i} = (s_{cs_i}, s_m)$$

where $EP(e_{cs_i})$ denotes the probability of invocation of m at cs_i.

► **Example.** In Fig. 1, the distribution of `this` in method `Circle.set` can be calculated as $1.0\bar{o_1}$, which is more precise than that calculated by using the formula in Section 3.3 (i.e., $0.25\bar{o_1} + 0.75\bar{o_2}$). ◄

The return value of a virtual invocation also needs to be taken into account in order to achieve a conservative resolution. Suppose a virtual invocation at a call site cs is

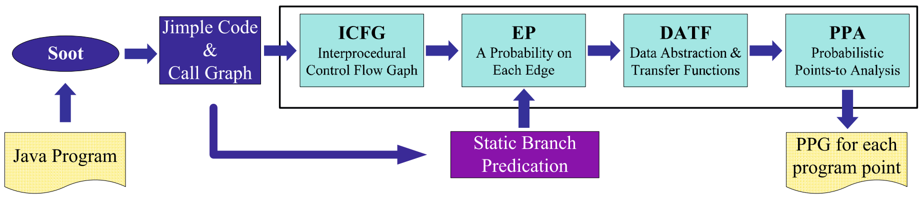

Soot
Jimple Code & Call Graph
ICFG Interprocedural Control Flow Gaph
EP A Probability on Each Edge
DATF Data Abstraction & Transfer Functions
PPA Probabilistic Points-to Analysis
Java Program
Static Branch Predication
PPG for each program point

Fig. 6. A framework of *Lukewarm* tool

$v = r.m()$. With the points-to set of r, we can deduce the set of methods that can be invoked at cs and denote each method remained in the set M_v. The distribution of the variable v can be calculated by

$$\Delta_v = \sum_{m_j \in M_v} EP(e) \cdot \Delta_{return_{m_j}}, \quad e = (s^{ret}_{m_j}, s'_{cs})$$

where $EP(e)$ denotes the probability of m_j invoked at cs, and $\Delta_{return_{m_j}}$ the distribution of the return value of m_j. $EP(e)$ is then adjusted on the base of the distribution of r.

▶ **Example.** In Fig. 1, the distribution of variable r is $0.25\bar{o_5} + 0.75\bar{o_6}$. ◀

4 Implementation

We have developed a tool called *Lukewarm* to support our method. Fig. 6 shows the framework of *Lukewarm*: the inputs are Jimple code (a typed 3-address *intermediate representation*) [17,18], the call graph of the program which are generated by Soot [19], and the predecessor-dependent probabilities computed by a static branch prediction analysis [16]; the outputs are the PPGs at all program points. *Lukewarm* provides engineers with support in probabilistic points-to analysis of Java programs by the following four steps:

- Use RTA algorithm in Soot to construct the call graph covering all reachable methods, and then construct the CFGs for these methods and combine them to form an ICFG;
- Annotate each edge with a probability calculated based on the static branch prediction approach;
- Extract all the references and objects from the ICFG and generate the transfer function for each node;
- Adopt a worklist algorithm on the nodes of the ICFG to compute the PPG for each program point. A worklist is initialized to contain a node which is associated with an empty PPG. Everytime one node n in the worklist is retrieved, and the PPGs of its G_{in} and G_{out} are calculated on the basis of the transfer function. If G_{in} or G_{out} of n is changed (The graphs G_a and G_b are equivalent *iff* $\|G_a - G_b\|_e < \epsilon$.), all the successive nodes of n are added to the worklist. This procedure is repeated until the worklist is empty. Based on the observations, we choose ϵ to be 0.01 in this paper for the reason that it can balance the efficiency and precise.

Table 2. Java benchmarks

Program	#Statement	#Block	Description
HashMap	20929	12307	A small program using HashMap in Java library.
ArrayList	150	27	A small program using ArrayList in Java library.
antlr	52058	23167	A parser generator and translator generator.(DaCapo)
xalan	24186	13716	An XSLT processor for transforming XML documents.(DaCapo)
luindex	24947	14144	A text indexing tool.(DaCapo)
hsqldb	24466	13714	An SQL relational database engine written in Java.(DaCapo)
toba-s	34374	16661	A tool translating Java class files into C source code.(Ashes)
Jtopas	32226	21042	A Java library for the common problem of parsing text data. (SIR)
JLex	32058	16120	A lexical analyzer generator for Java.
java_cup	37634	18049	A LALR parser generator written in Java.

5 Experiments

We have conducted three experiments on JPPA by comparing it with a context-insensitive and flow-sensitive points-to analysis (TPA for short) proposed by Hind et al [20]. The principle of TPA is to use an iterative dataflow analysis framework to compute the points-to set for each reference variable on each node of the CFG of the program. The first experiment was conducted to evaluate the precision of points-to sets calculated through JPPA, the second one was to evaluate the capability of JPPA to calculate the points-to relations *maybe* holding, and the third one was to evaluate the performance of JPPA. All experiments were conducted on a machine with an AMD Sempron^{TM} 1.80GHz CPU and 1G heap size (option -Xmx1024m).

Table 2 shows benchmark programs used in the experiments, which include the SIR suite [21], programs from the Ashes suite [22], programs from the DaCapo suite [23], and two programs for testing `java.util.HashMap` and `java.util.ArrayList`. Table 2 also shows the total number of statements (#Statement) in Jimple code and that of basic blocks (#Block) of each benchmark program.

5.1 Precision of Points-to Analysis

In the first experiment, the points-to sets computed by JPPA and TPA for each reference were the same at each program point. Table 3 shows the average and the maximum sizes of the points-to sets of each benchmark program as well the percentage of the points-to set with only one object inside. It can be seen that for each benchmark program the average size of points-to sets is less than 2, and the maximum size is less than 20. A rational claim is that the closer to 1 the average size of points-to sets is, the more traditional optimization techniques we can use because optimizations usually require the information about the points-to relations *definitely* holding. For example, if the points-to set of the receiver variable contains only one object, it can reduce the direct overhead of dispatching the message and provide the opportunities for method inlining and interprocedural analysis.

In addition, most points-to sets of the benchmark programs have only one object inside, as the row OneObject.Pt (%) indicates. It means that both JPPA and TPA can explore most points-to relations.

Table 3. JPPA measurements

Program	HashMap	ArrayList	antlr	xalan	luindex	hsqldb	toba-s	Jtopas	JLex	java_cup
#Avg.Size.Pt	1.49	1.32	1.05	1.60	1.95	1.49	1.29	1.86	1.04	1.72
#Max.Size.Pt	6	8	13	13	13	13	19	13	9	18
OneObject.Pt(%)	79.57	93.80	98.50	86.22	77.58	81.34	91.33	74.12	99.15	87.58

5.2 Precision of Probabilities

In the second experiment, we chose some program points. For each program point l, we obtained the corresponding dynamic probabilistic points-to graph G_d as a standard and then evaluated JPPA and TPA by calculating their probabilistic points-to graphs at l (say G_s^{JPPA} and G_s^{TPA}, respectively) and the corresponding average graph distances to G_d. In this experiment, G_d was computed by using a dynamic method:

– Perform a program instrumentation in order to collect at runtime the hashcodes of all objects in the program and variables at l;
– Execute the program and use the hashcodes of the objects to explore the points-to relations at l;
– Execute the corresponding benchmark program multiple times and record the frequency for each points-to relation.

The average of distance can be calculated by

$$AVG_{distance} = \frac{\sum \|G_s - G_d\|_e}{N}$$

where N is the number of program points of interest, and G_s is a probability points-to graph of any program point generated either by JPPA or TPA, and $\|G_s - G_d\|_e$ calculates the average of the normalized Euclidean distances between the distributions of G_s and G_d, which can be calculated by

$$\|G_s - G_d\|_e = \frac{\sum\limits_{r \in R} \sqrt{0.5 \cdot \sum\limits_{o \in Obj} (\Delta_r^s(o) - \Delta_r^d(o))^2}}{|R|}.$$

$AVG_{distance}$ ranges from zero to one and was used to measure the divergence between a probability points-to graph computed by JPPA or TPA and that computed by the dynamic method. $AVG_{distance} = 0$ means that the PPG computed by JPPA or TPA is regarded as correct, and $AVG_{distance} = 1$ means that it may be totally wrong.

In the experiment, when using JPPA, we assumed that all incoming edges of a node of ICFG have the same probability, and then computed the probabilistic points-to relations. When using TPA, we assumed that the probability of each points-to relation belonging to a points-to set is equal. Fig. 7 shows the average distances between the PPGs computed by JPPA and TPA and those computed by the dynamic method. For each benchmark program, the average distance corresponding to JPPA is shorter than that corresponding to TPA. However, for the programs **hsqldb** and **Jtopas**, the distances corresponding to JPPA are not of significant divergences to those corresponding

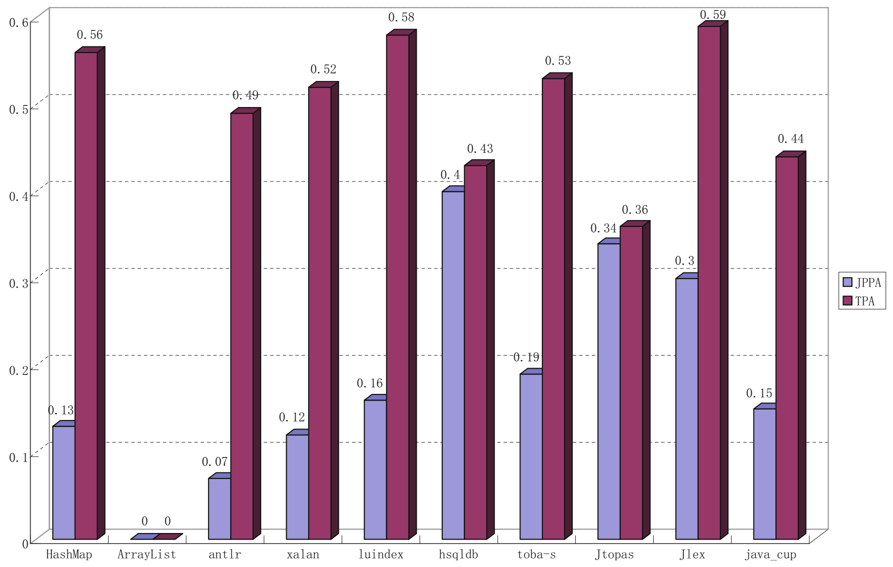


Fig. 7. The comparison of graph distances between JPPA and TPA

to TPA. After a thorough investigation, we found that the arrays of objects used in these programs can decrease the precision of JPPA to compute probabilities. It is one of our future research directions to find out how to improve the precision of JPPA when a number of arrays of objects are available in the programs.

JPPA reaches the same precision of points-to sets as that of TPA, but has a much higher precision of probabilities than that of TPA. In addition, the precision of probabilities of JPPA is very close to that of the dynamic method. It demonstrates that JPPA can be an effective approach to probabilistic points-to analysis of Java programs.

5.3 Analysis Performance

In the third experiment, an investigation of the execution time and memory usage was conducted after the benchmark programs were executed several times. Table 4 shows the running time and memory usage for each benchmark program. A vertical line | is used to divide a result of using JPPA from that of using TPA. We also counted the number of the references (#reference), static fields (#static-field), and objects (#object) for each benchmark program in order to find out the factors that will affect the performance.

After analyzing the results, we found that the distributions of the static fields significantly affects the performance of both JPPA and TPA because the points-to sets of these fields can propagate to all nodes in the ICFG. Specially, when a points-to set of a static field is of a large number of elements, the performance will decrease rapidly because of the propagation of a great amount of data. In addition, the use of arrays of objects in Java also decrease the performance of JPPA because they are handled conservatively and thus a number of redundant objects are contained in points-to sets and also will be propagated.

Table 4. Performance

Program	#reference	#static-field	#object	Time (Sec)		Memory (MB)	
HashMap	4200	104	296	32.98	33.47	176.36	195.82
ArrayList	42	0	9	1.41	1.36	0.38	0.31
antlr	8651	158	972	329.03	369.52	239.79	203.98
xalan	4906	133	398	36.54	40.98	206.56	228.24
luindex	5095	133	419	43.06	43.87	218.97	237.01
hsqldb	4883	114	417	41.26	41.03	213.33	227.99
toba-s	5876	140	750	131.63	103.96	316.97	331.69
Jtopas	5896	107	440	139.70	167.45	298.07	330.80
JLex	5491	129	481	56.90	54.23	296.81	306.58
java_cup	6431	132	825	215.14	146.64	303.02	334.37

From the experiment, we found that JPPA spends more time and memory in computing the probabilities for the points-to relations than TPA does. However, the cost of using JPPA is only about 1.2 times of that of using TPA, which is still acceptable.

5.4 Threats to Validity

A threat to validity is that when using JPPA, we assumed that all incoming edges of a node in the ICFG have the same probability, and when using TPA, we assumed that the probability of each points-to relation belonging to a points-to set is equal. In practice, these assumptions are very strong in that edges of ICFG may hold different probabilities, which may be hardly determined precisely before the analysis. A more reasonable solution is to use the edge profiling information estimated by the machine learning techniques to compute the predecessor-dependent probabilities which will be explored in our future work.

6 Related Work

In the past several years, points-to analysis has been an active research field. A survey of algorithms and metrics for points-to analysis has been given by Hind [24].

Traditional points-to analysis. Context-sensitivity and flow-sensitivity are two major dimensions of pointer analysis precision. Context-sensitive and flow-sensitive algorithms (CSFS) [25,26,27,28] are usually precise, but are difficult to scale to large programs. Recently Yu et al. [29] proposed a level-by-level algorithm that improves the scalability of the context-sensitive and flow-sensitive algorithm. Context-insensitive and flow-insensitive (CIFI) algorithms [30,31] have the best scalability on the large programs with overly conservative results. Equality-based analysis and inclusion-based analyses have become the two widely accepted analysis styles. Through carrying out the experiments, Foster et al. [32] compared several variations of flow-insensitive points-to analysis for C, including polymorphic versus monomorphic and equality-based versus inclusion-based.

How well the algorithms scale to large programs is an important issue. Trade-offs are made between efficiency and precision by various points-to analyses, including context-sensitive and flow-insensitive analyses [4,5,33] and context-insensitive and flow-sensitive analyses [6,20,7]. However, these conventional points-to analyses do not provide with the probabilities for the possible points-to relations, which is one of main goals of JPPA proposed in this paper.

Probabilistic pointer analysis for C. With the proposition of the speculative optimizations, the probability theory has been introduced into the traditional program analysis. In earlier work, Ramalingam [15] proposed a generic data flow frequency analysis framework that uses the edge frequencies propagation to compute the probability a fact holds true at every control flow node. Chen et al. [8] developed a context-sensitive and flow-sensitive probabilistic point-to analysis algorithm. This algorithm is based on an iterative data flow analysis framework, which computes the transfer function for each control flow node and propagates probabilistic information additionally. In addition, this algorithm also handles interprocedural points-to analysis on the basis of Emami's algorithm [25]. Their experimental results demonstrates that their technique can estimate the probabilities of points-to relations in benchmark programs with reasonable accuracy although they have not disambiguated heap and array elements. Compared with Chen et al.'s algorithm, JPPA is also on the basis of an iterative data flow analysis framework but with context-insensitive analysis. In addition, the concept of the discrete probability distribution are introduced to JPPA.

Silva and Steffan [34] proposed a one-level context-sensitive and flow-sensitive probabilistic pointer analysis algorithm that statically predicts the probability of each points-to relation at every program point. Their algorithm computes points-to probabilities through the use of linear transfer functions that are efficiently encoded as sparse matrices. Their experimental results demonstrates that their analysis can provide accurate probabilities, but omits handling alias between the shadow variables. JPPA provides a safer result in its analysis because the transfer functions of accessing of instance fields do not use the shadow variables during its propagation of the distributions.

7 Conclusions

In this paper we have presented JPPA, a context-insensitive and flow-sensitive algorithm for calculating the probabilistic points-to relations in Java programs. JPPA also predicts the likelihood of points-to relations without depending on runtime profiles. In order to ensure the safety of the result, JPPA takes Java libraries into account. We have conducted experiments to validate JPPA by comparing it with the traditional context-insensitive and flow-sensitive points-to analysis. The experimental results show that JPPA not only produces precise probabilities of points-to relations for Java, but also maintains a similar performance to the traditional context-insensitive and flow-sensitive points-to analysis.

In the future, we would like to continue our efforts to improve and extend JPPA to support more powerful functions concerned with points-to analysis, such as handling of arrays of objects. Also with a more mature approach and the corresponding tool, we would like to apply them in some large-scale projects to investigate how our approach can benefit real projects. We would also like to extend JPPA to a context-sensitive analysis approach in order to improve its precision.

Acknowledgements

We would like to thank the anonymous reviewers for their useful comments and suggestions. We would also like to thank Longwen Lu, Hongshen Zhang, Yu Kuai and Cheng Zhang for their discussions on this work. This work was supported in part by National Natural Science Foundation of China (NSFC) (Grant No. 60970009 and 60673120).

References

1. Das, M., Liblit, B., Fähndrich, M., Rehof, J.: Estimating the impact of scalable pointer analysis on optimization. In: Proceedings of the 8th International Static Analysis Symposium, pp. 260–278 (2001)
2. Hind, M., Pioli, A.: Which pointer analysis should I use? In: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 113–123 (2000)
3. Chatterjee, R., Ryder, B.G., Landi, W.A.: Complexity of points-to analysis of Java in the presence of exceptions. IEEE Transactions on Software Engineering 27(6), 481–512 (2001)
4. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for Java. ACM Transactions on Software Engineering and Methodology 14, 1–41 (2002)
5. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, pp. 131–144 (2004)
6. Choi, J.D., Burke, M., Carini, P.: Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 232–245 (1993)
7. Hardekopf, B., Lin, C.: Semi-sparse flow-sensitive pointer analysis. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 226–238 (2009)
8. Chen, P.S., Hwang, Y.S., Ju, R.D.C., Lee, J.K.: Interprocedural probabilistic pointer analysis. IEEE Transactions on Parallel and Distributed Systems 15(10), 893–907 (2004)
9. Oancea, C.E., Mycroft, A., Harris, T.: A lightweight in-place implementation for software thread-level speculation. In: Proceedings of the 21st Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 223–232 (2009)
10. Scholz, B., Horspool, R.N., Knoop, J.: Optimizing for space and time usage with speculative partial redundancy elimination. In: Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, pp. 221–230 (2004)
11. Dai, X., Zhai, A., chung Hsu, W., chung Yew, P.: A general compiler framework for speculative optimizations using data speculative code motion. In: Proceedings of the 2005 International Symposium on Code Generation and Optimization, pp. 280–290 (2005)
12. Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static class hierarchy analysis. In: Proceedings of the 9th European Conference on Object-Oriented Programming, pp. 77–101 (1995)
13. Bacon, D.F., Sweeney, P.F.: Fast static analysis of C++ virtual function calls. In: Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, pp. 324–341 (1996)
14. Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E., Godin, C.: Practical virtual method call resolution for Java. In: Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 264–280 (2000)

15. Ramalingam, G.: Data flow frequency analysis. In: Proceedings of the 1996 Conference on Programming Language Design and Implementation, pp. 267–277 (1996)
16. Wu, Y., Larus, J.R.: Static branch frequency and program profile analysis. In: Proceedings of the 27th Annual International Symposium on Microarchitecture, pp. 1–11 (1994)
17. Vallée-Rai, R., Hendren, L.J.: Jimple: simplifying Java bytecode for analyses and transformations. Sable technical report, McGill (1998)
18. Vallée-Rai, R.: The Jimple framework. Sable technical report, McGill (1998)
19. Soot: http://www.sable.mcgill.ca/soot
20. Hind, M., Burke, M., Carini, P., deok Choi, J.: Interprocedural pointer alias analysis. ACM Transactions on Programming Languages and Systems 21(4), 848–894 (1999)
21. Do, H., Elbaum, S., Rothermel, G.: Infrastructure support for controlled experimentation with software testing and regression testing techniques. Empirical Software Engineering: An International Journal 10, 405–435 (2004)
22. Ashes Suite Collection, http://www.sable.mcgill.ca/software
23. Blackburn, S.M., Garner, R., Hoffman, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications, pp. 169–190 (2006)
24. Hind, M.: Pointer analysis: haven't we solved this problem yet? In: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, pp. 54–61 (2001)
25. Emami, M., Ghiya, R., Hendren, L.J.: Context-sensitive interprocedural points-to analysis in the presence of function pointers. In: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, pp. 242–256 (1994)
26. Landi, W., Ryder, B.G., Zhang, S.: Interprocedural modification side effect analysis with pointer aliasing. In: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation, pp. 56–67 (1993)
27. Whaley, J., Rinard, M.: Compositional pointer and escape analysis for Java programs. In: Proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, pp. 187–206 (1999)
28. Wilson, R., Lam, M.S.: Efficient context-sensitive pointer analysis for C programs. In: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation, pp. 1–12 (1995)
29. Yu, H., Xue, J., Huo, W., Feng, X., Zhang, Z.: Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In: Proceedings of the 8th International Symposium on Code Generation and Optimization, pp. 218–229 (2010)
30. Steensgaard, B.: Points-to analysis in almost linear time. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 32–41 (1996)
31. Andersen, L.: Program analysis and specialization for the C programming language. DIKU report 94-19, University of Copenhagen (1994)
32. Foster, J.S., Fähndrich, M., Aiken, A.: Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In: SAS 2000. LNCS, vol. 1824, pp. 175–199. Springer, Heidelberg (2000)
33. Lhoták, O., Hendren, L.: Context-sensitive points-to analysis: Is it worth it? In: Mycroft, A., Zeller, A. (eds.) CC 2006. LNCS, vol. 3923, pp. 47–64. Springer, Heidelberg (2006)
34. Silva, J.D., Steffan, J.G.: A probabilistic pointer analysis for speculative optimizations. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 416–425 (2006)