Probabilistic Points-to Analysis for Java

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

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

Abstract. Points-to analysis, widely used in program analysis and compiler optimizations, is an analysis technique for computing a relation between variables of pointer types and their allocation sites. It is commonly used to reason the *may/-must points-to sets* of the reference variables, while it may not be easily used in speculative optimizations because they usually require quantitative information on the likelihood of the points-to relation. Although there are some probabilistic points-to analyses for C language, there is no probabilistic points-to analysis for Java so far. In this paper, we propose a *context-insensitive and flow-sensitive* (CIFS) *probabilistic points-to analysis for Java* (JPPA) that is used to statically predict the probability of each points-to relation at each program point for Java programs. We also conducted an experiment to compare JPPA with a traditional CIFS points-to analysis approach in order to demonstrate that JPPA provides an accurate resolution.

Keywords: points-to analysis, probability, Java

1 Introduction

Points-to analysis is a useful technique which is widely used in compiler optimization and program analysis tasks [1, 2]. The goal of points-to analysis is to compute a points-to relation between variables of pointer types and allocation sites. Context-sensitivity and flow-sensitivity are two major aspects of points-to analysis for improving the precision of the analysis, which is extremely important for compiler optimization and program analysis. Context-sensitive points-to analysis [3, 4] distinguishes among the different contexts under which a method is invoked, and analyzes the method separately for each context. Flow-sensitive points-to analysis [5, 6] takes into account the flow of control between program points inside a method, and computes separate solutions for these points. Specially, in flow-sensitive analysis, points-to analysis can be used to deduce that for each points-to relation whether it definitely exists, or maybe exists at any program point. However, for the cases that a points-to relation maybe exists, the conventional points-to analysis cannot provide the quantitative information of the likelihood of the points-to relation holding.

Probabilistic points-to analysis, which defines the probability of each points-to relation, becomes a promising technique in speculative optimizations for providing the compiler with more optimization chances. Although there are some probabilistic points-to analysis for C language, there is no probabilistic points-to analysis for Java so far.

Furthermore, there is a big challenge how to compute the probability reflecting the likelihood of the points-to relation in the real program execution at an acceptable cost. In this paper, we propose a static *context-insensitive and flow-sensitive* (CIFS) *Probabilistic Points-to Analysis for Java* (JPPA) that is used to statically predict the probability of each points-to relation at each program point for Java programs. JPPA extends the traditional points-to analysis by introducing the probability. In JPPA, the points-to relationship is not confined to yes/no but is associated with a real number representing the likelihood, which enlarges the application of the points-to analysis. JPPA uses a novel algorithm based on data-flow analysis which can propagate points-to relations efficiently. Our analysis does not rely on the runtime profiling information of the program and can optionally use edge profiling. In addition, we have developed a JPPA tool that implements a static probabilistic CIFS points-to analysis, which is safe, accurate, and can scale to the benchmark programs with Java library code.

The algorithm proposed in this paper is expected to benefit the speculative optimization [7–9]. A speculative optimization is a code transformation which allows ambiguous memory access in a potentially unsafe order and requires a recovery mechanism to ensure program correctness. Especially, speculative multithreading (SpMT) [10, 11], an optimization technique for achieving faster execution of sequential programs on multiprocessor hardware, can benefit from the probabilistic points-to analysis algorithm. In SpMT architecture, sequential program is divided into threads running in the processor. When two threads lead to data dependence violation, the recovery mechanism is triggered to ensure the correctness of sequential semantics. Although hardware provide the recovery mechanism, the compiler needs to estimate the likelihood of dependence so that it can enlarge the number of parallelable threads and reduce the number of threads violating dependence. Therefore, probabilistic points-to analysis can be a significant technology to guide the extraction of the threads from the sequential programs in the speculative multithreading model.

This paper makes the following contributions:

1. We propose a novel static probabilistic points-to analysis algorithm for Java which extends the traditional context-insensitive and flow-sensitive points-to analysis by introducing the probability;
2. We provide JPPA, a probabilistic points-to analysis for Java. JPPA is based on data-flow analysis technique and also takes into account the main object-oriented features, for instance inheritance and polymorphism;
3. We have developed a JPPA tool and conducted an experiment using JPPA tool to analyze nearly 24K lines of Java code within 2 minutes on average and obtain the average points-to set size with 1.5.

The remainder of the paper is organized as follows. Section 2 presents an example to illustrate how the analysis algorithm works. Section 3 presents the algorithm for probabilistic points-to analysis from intraprocedure to interprocedure. Section 4 describes the implementation of JPPA. Section 5 presents our experimental results. Section 6 compares the related work that has been compared in. Section 7 concludes this paper.

2 Example

We next use an example to illustrate how our analysis works. Fig. 1 shows a Java program that contains four classes: `Shape`, `Round`, `Square` and `Main`. Classes `Round` and `Square` extend class `Shape`. In the sample program, the objects created are identified by the comments with the form `//oi (i=1,2,3,4,5,6,7)`. The method `set` assigns the objects to the fields of the classes `Shape`, `Round` and `Square`. The method `randomRealNum` simulates the actual input data. The method `Main` is the entry method that contains **if-else** branch, sequence and **loop** structure. And it contains four reference variables `p`, `q`, `r` and `t` and an array type variable `list`.

There is a field `area` in class `Shape` and a shadow declaration of the field `area` in the class `Round`. In order to distinguish these two fields, the field declared in class `Shape` is marked as `Shape.area`. The problem does not happen in class `Square`, because it just inherits `area` from class `Shape`.

```
1  public class Shape {
2       public Double area = null;
3       public Double set(Double s) {
4            this.area = s;
5            return s;
6       }
7  }
8  public class Square extends Shape {
9       public Double sideLength = null;
10      public Double set(Double s) {
11           this.area = s;
12           this.sideLength =
               new Double(Math.sqrt(s));      //o6
13           return this.sideLength;
14      }
15 }
16 public class Round extends Shape {
17      public Double area = null;
18      public Double radius = null;
19      public Double set(Double s) {
20           super.area = new Double(0);      //o4
21           this.area = s;
22           this.radius =
               new Double(Math.sqrt(s/3.14)); //o5
23           return this.radius;
24      }
25 }
26 public class Main {
27      public static void main(String args[]) {
28           int a = randomInt();
29           int b = randomInt();
30           Shape p = null;
31           if(a>0 && b>0) {
32                p = new Round();            //o1
33           }
34           else {
35                p = new Square();           //o2
36           }
37           Double q =
                  new Double(Math.abs(a*b)); //o3
38           Double r = p.set(q);
39           Double [] list = new Double[2];
40           list[0] = new Double(9);         //o7
41           list[1] = q;
42           int i = 0;
43           while(i<list.length)
44           {
45                Double t = null;
46                t = list[i];
47                p.area = t;
48                i = i+1;
49           }
50      }
51      private static double randomInt() {
52           return Math.floor(Math.
                       random()*11-5);
53      }
54 }
```

Fig. 1. A Java program.

In this simple case, we assume that `randomRealNum` generates the real numbers in the range from -5 to +5 complying to the uniform distribution. The probabilities of the **if-else** branch can easily be referred in the method `main` that the `if` statement in line 31 is taken with probability 0.25 and the `else` statement in line 34 is taken with a probability of 0.75. In line 38, the method `set` is called and the return value is assigned to the variable `r`. The code from line 39 to 41 is the operation on the array type variable. The loop structure is demonstrated from line 43 to 49. In the following, we start to illustrate our analysis with the example.

There are four steps of our analysis procedure. At the beginning, the analysis scope should be established. Call graph (CG) is used to reduce the unreachable methods. The right part of Fig. 2 illustrates the call graph of the program in Fig. 1 built using the points-to information provided by the traditional CIFS points-to analysis.

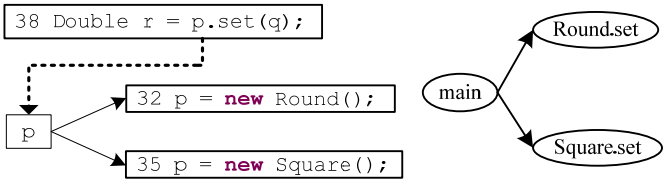
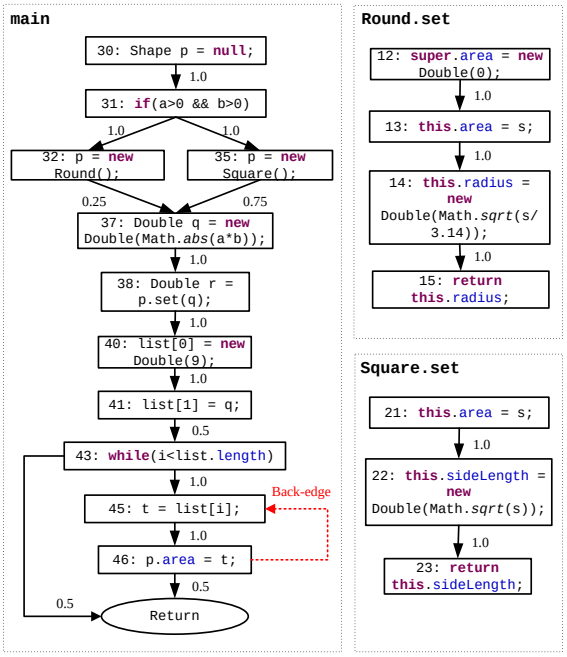


Fig. 2. The call graph of the program in Fig. 1.

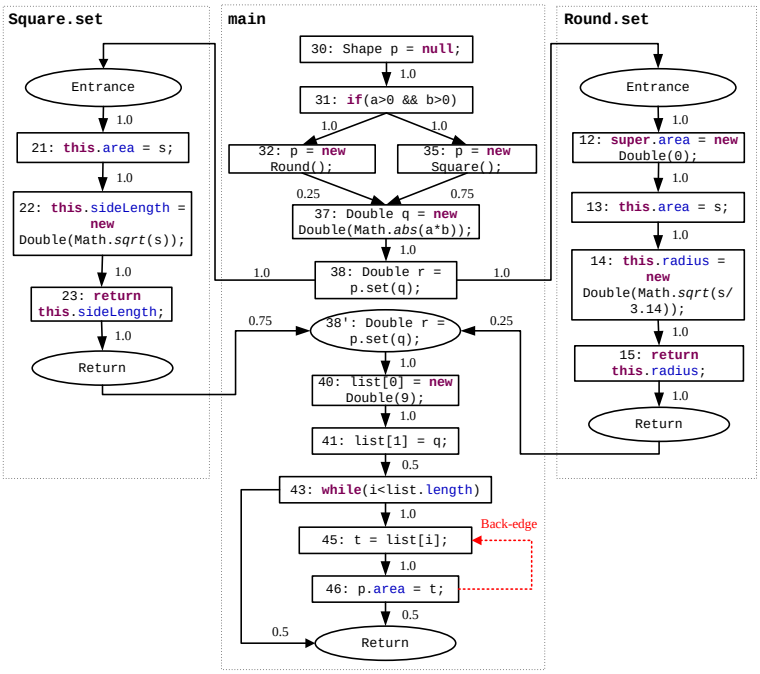
The second step is to construct intraprocedural control flow graph for each method in the call graph. Fig. 3(a) shows the control flow graphs (CFG) of method `main`, `Round.set` and `Square.set` respectively. Each node corresponds to a statement in the sample program with the line number. Each edge is labelled with a probability from the source node to the destination node. The back-edge in the loop structure is identified.

The third step is to combine the CFGs of the methods being analyzed into one interprocedural CFG (ICFG). The probability on the interprocedural edges are adjusted according to the probability of the receiver object during the whole analysis. Fig. 3(b) reveals the ICFG with the stable probability on the interprocedural edges. There are additional nodes generated in the ICFG. In line 38, there is method invocation. Line 38' node is the copy of the node labelled with line 38, which is used for receiving return value. In the method `Square.set`, **Entrance** node assigns the actual parameters to the formal parameters and **Return** node makes the return value unique.

The final step is to analyze each statement on the ICFG. Fig. 4 illustrates the probabilistic points-to analysis from line 27 to line 37 in the method `main`. The probabilistic points-to graphs (a) and (b) are calculated after analyzing lines 32 and 35. Before the line 37, graph (c) is generated by merging (a) and (b). The graph (d) is the analysis result after analyzing line 37. Because of the polymorphism in line 38, the methods named `set` both in classes `Round` and `Square` are involved in the invocation site. The probabilities on the return edges from `Square.set` and to `main` is computed according to the points-to possibilities in the points-to set of the variable `p`. For example, the probability of `p` pointing to object `o1` is 0.25 and the type of `o1` is `Round`, it can be inferred that the probability of the return edge from `Round.set` to `main` is 0.25. Fig. 5 manifests the analysis in `Round.set` and `Square.set`. Fig. 6 shows the analysis from the end of method call to the end of method `main`. The graph (l) is built by merging the graph (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 using the graph (l) and replacing the return node by `r` node. The graphs (n),(o),(p) and (r) are



(a) CFGs



(b) ICFG

Fig. 3. The CFGs and ICFG of the program in Fig. 1.

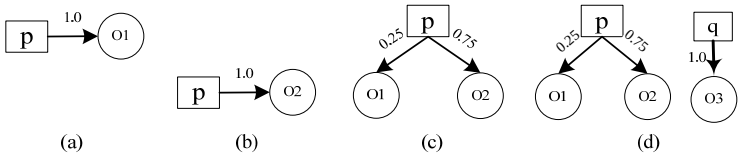


Fig. 4. The probabilistic points-to analysis for method `main` (part I).

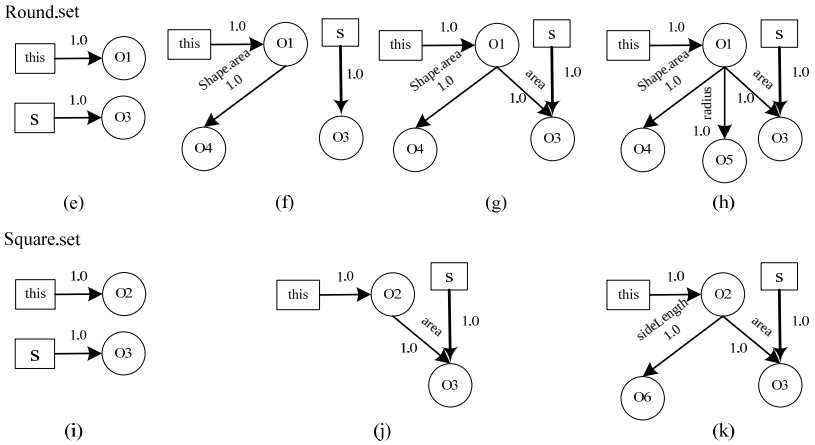


Fig. 5. The probabilistic points-to analysis for methods.

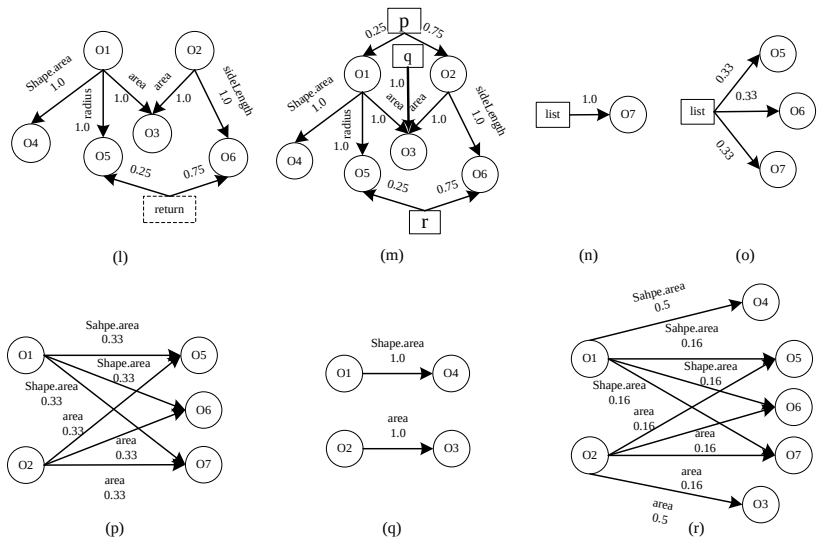


Fig. 6. The probabilistic points-to analysis for method `main` (part II).

the incremental or change parts of the graph (m) because of the clarity. The graphs (n) and (o) represent the analysis after lines 40 and 41 respectively. There are two situations taken into account when handling the loop. The first situation is to enter the loop with infinite time. The loop body ranged by *back-edge* is unfolded repetitively until the analysis result dose not change. The graph (p) illustrates the result of this situation. The second situation is not entering the loop. The graph (q) as a part of (m) manifests this situation. Considering these two situations comprehensively, the graph (r) is calculated by merging the graphs (p) and (q) with equal probability.

3 Probabilistic Points-to Analysis

We next present our probabilistic points-to analysis for Java. Section 3.1 describes the probabilistic points-to graph and the program representation. Section 3.2 represents the intraprocedural probabilistic points-to analysis. Section 3.3 extends the intraprocedural analysis to interprocedural analysis Section 3.4 discusses the safety about our analysis.

3.1 Probabilistic Points-to Graph and Program Representation

We first introduce the conventional points-to graph representation [3]. To perform points-to analysis for Java programs, three sets are defined. The first is set Ref which contains all reference variables in the analyzed program and also includes the static fields in classes. The second is set Obj which contains the names of all objects that are created at object allocation sites; for each allocation site s_i, object name $o_i \in Obj$ is unique in whole program. The third is set $Field$ which contains instance fields in the classes, but not includes the static fields. There are two types of edges in the points-to graphs. Edge $(r, o_i) \in Ref \times Obj$ represents that reference variable r points to object o_i and Edge $(\langle o_i, f\rangle, o_j) \in (Obj \times Field) \times Obj$ represents that field f of object o_i points to object o_j.

Let *d* represent the points-to relationship. The form of *d* can be either (r, o_i) or $(\langle o_i, f\rangle, o_j)$. The goal of probabilistic points-to analysis is to compute the probability of each points-to relation holding at every program point. The expected probability is defined by the following function

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

where *Expected(s)* denotes the number of times a program point *s* is expected to turn up on program execution path and *Expected(s,d)* denotes the number of times relation *d* is expected to hold dynamically at program point *s* [12].

The discrete probability distribution (distribution for short) is introduced into our analysis framework, because it is convenient to compare the probability of points-to relation within one points-to set. A discrete probability distribution over a set O is a mapping

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

The *support* of Δ is given by

$$\lceil \Delta \rceil := \{o \in O | \Delta(o) > 0\}.$$

$\bar{o}$ denotes the point distribution, satisfying

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

If Δ_i is a distribution for each i in some finite index set I and $\sum_{i \in I} p_i = 1$, $\sum_{i \in I} p_i \cdot \Delta_i$ is also a distribution and given by

$$(\sum_{i \in I} p_i \cdot \Delta_i)(o) = \sum_{i \in I} p_i \cdot \Delta_i(o), o \in O$$

Every distribution can be represented as a linear combination of the point distributions and can be written in the following form

$$\Delta = \sum_{o \in \lceil \Delta \rceil} \Delta(o) \cdot \bar{o}$$

For a fixed program point *s*, the points-to set of a heap reference r or $\langle o_i, f \rangle$ with probability can be seen as a probability distribution over object set Obj. The probabilistic points-to relationships at that point can be seen as a set of the distributions.

For example, the distribution of variable p before line 38 can be represented as follows:

$$\Delta_p = 0.25\bar{o_1} + 0.75\bar{o_2}$$

The *probabilistic points-to graph* (PPG) is a directed multi-graph which has the same types of nodes with the conventional one and contains two kinds of edges a little different from that. Each edge has a probability of points-to relation keeping. The directed edges only with probability label which represents the points-to relationship from a variable to an object. The edges from an object to another means the field of one object points to another object and these edges has both a class field label and a probability label.

The probabilistic points-to graph can be denoted as a distribution set of the reference variables. For example, at the program point after line 38 in Fig.1, the probabilistic points-to graph are the distributions of p, q, r, and object field references.

$$\Delta_p = 0.25\bar{o_1} + 0.75\bar{o_2} \qquad \Delta_{o1.radius} = 1.0\bar{o_5}$$

$$\Delta_r = 0.25\bar{o_5} + 0.75\bar{o_6} \qquad \Delta_{o1.area} = 1.0\bar{o_3}$$

$$\Delta_q = 1.0\bar{o_3} \qquad \Delta_{o1.Shape.area} = 1.0\bar{o_4}$$

$$\Delta_{o2.area} = 1.0\bar{o_3} \qquad \Delta_{o2.sideLength} = 1.0\bar{o_6}$$

Program is represented by the ICFG whose edges are labelled with a probability. For the case that the edges share one destination node with different source nodes, the probability of each edge can be computed as follows

$$Prob(e_i) = \frac{Freq(e_i)}{\sum_{j \in I} Freq(e_j)}$$

where $e_i, i \in I$ are all edges which share a certain destination node, $Freq(e_i)$ is the execution frequency of edge e_i. These probabilities can be computed by either static or dynamic way.

In Java there are four basic ways to assign a value to a reference variable which may change the points-to relation:

1. Object creation: $v = new\ C$
2. Direct assignment: $v = r$
3. Instance field read: $v = r.f$
4. Instance field write: $v.f = r$

3.2 Intraprocedural Analysis

The conventional points-to analysis can be formulated as a data flow framework [5]. The data flow framework includes transfer functions which formulate the effect of statements on points-to relations. Either a statement or a block (i.e. a sequence of the statements) can be seen as a node in the CFG. The next discussion based on that one statement corresponds one node in the CFG.

A monotonic dataflow analysis problem is a tuple $(L, \sqcup, Fun, P, Q, E, \iota, M)$, where:

- $\sqcup$ is the meet operator
- $Fun \subseteq L \mapsto L$ is a set of monotonic functions from L to L
- P is the set of the statements labels
- Q is the set of flows between statement labels which is the subset of $\{(l, l')|l, l' \in P\}$
- E is the initial set of statements labels
- ι specifies the initial analysis information
- $M : E \mapsto F$ is a map from control flow edges to transfer functions

Under the conventional points-to analysis scenario, a points-to graph can be regarded as an element of a lattice L. The partial order over L is determined by the points-to set inclusion relation of each variable in graph. And the meet operation of the two graphs is to union the points-to set for each variable.

Let l be the label of a statement s, $G_{in}(l)$ and $G_{out}(l)$ represent the points-to graph at the program points before and after the statement s respectively. The statement labelled with l is associated with the transfer function that transforms $G_{in}(l)$ to $G_{out}(l)$. The computations of $G_{in}(l)$ and $G_{out}(l)$ are as follows:

$$G_{in}(l) = \begin{cases} \iota & if\ l \in E \\ \bigsqcup\{G_{out}(l')|(l', l) \in Q\} & otherwise \end{cases}$$
$$G_{out}(l) = f_l(G_{in}(l))$$

where $f_l \in F$ is the transfer function of the statement labelled l. The analysis iteratively computes the $G_{in}(l)$ and $G_{out}(l)$ for all nodes until convergence. The transfer function of a block can be easily lifted by the composition of the transfer functions of the statements within the block.

The probabilistic points-to analysis can be formulated as a data flow framework as well. In our analysis, E only contains the label of the first statement during the program execution and ι is a special probabilistic points-to graph which all the reference variables point to undefined target (i.e. UND) with total probability.

Basic Points-to Assignment Statements For each kind of statement l, there is a transfer function F_l corresponding to it. F_l takes $G_{in}(l)$ as input and computes the result $G_{out}(l)$. The transfer functions first copy $G_{in}(l)$ to $G_{out}(l)$ and then update $G_{out}(l)$ using the data in $G_{in}(l)$ or not. Fig. 7 describes these transfer functions.

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

Fig. 7. Computing distributions.

The first case is simple. Transfer function updates the distribution $\Delta_v^{G_{out}(l)}$ with the point distribution of o creating by $new\ C$ expression. In the second case, the distribution $\Delta_v^{G_{out}(l)}$ is replaced by the distribution $\Delta_r^{G_{in}(l)}$. The third case deals with the field access by considering the distribution composition of the field f of all the object in the support set of $\Delta_r^{G_{in}(l)}$. These three cases are strong updating which change the left distribution by the right part entirely. The last case is a weak updating, because the points-to relation before the statement execution has been reserved according to some probability. Moreover, it also updates multiple distributions in the form $o.f$.

In Java language, array provides the most efficient way to store and access object reference sequence randomly. Since the usage of array in Java is common, probabilistic points-to analysis should take array into account. An array can be regarded as a multiple reference variable. Since every array is in the constructor $new\ C\ [n]$ and n can be either a constant or a variable, it is difficult to infer the range of n in our analysis framework. A heuristic is used to approximate how many references are represented by a given multiple reference variable if this information cannot be determined statically. The initialization of an array can be formulated as follows:

$$C\ [\,]\ array\ =\ new\ C\ [N]$$

When accessing the element of $array$ by this way $array[i] = obj$, obj is added to the points-to set of $array$ (the points-to set of $array$ can be represented by $Pt(array)$)

and each points-to probability is recalculated by the following equation:

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

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

For example, at the program point after line 41 in Fig. 1, the distribution of the variable `list` is computed by two steps. First, the points-to set is calculated.

$$Pt(list) = \lceil \Delta_q \rceil \cup \{o_7\} = \{o_5, o_6, o_7\}$$

And then distribution is computed.

$$\Delta_{list} = \frac{1}{3}\bar{o_5} + \frac{1}{3}\bar{o_6} + \frac{1}{3}\bar{o_7}$$

Branch When the multiple nodes directly reach the destination node in the CFG, the meet operation is involved in the calculation of the PTG incoming the destination node. Compared with the meet operation in the traditional points-to analysis, it is changed due to the probability being introduced.

Suppose the program point l is the successor of the program points $l_i (i \in I)$. The following condition is satisfied

$$\sum_{i \in I} Prob((l_i, l)) = 1$$

where $Prob((l_i, l))$ represents the probability from program point l_i to l when execution. The meet operation can be described by the following equation.

$$\begin{aligned} G_{in}(l) &= \bigsqcup \{G_{out}(l_i) | i \in I\} \\ &\triangleq \sum_{i \in I} Prob((l_i, l)) \cdot G_{out}(l_i) \end{aligned}$$

The most commonly used conditional is the ***if-then-else*** construct. Suppose $G_{out}(l_{then})$ and $G_{out}(l_{else})$ are the sets of points-to relations at the exit points of ***then*** and ***else*** branches respectively, while p_t and p_f are the branching probabilities of ***then*** and ***else*** branches respectively and $p_t + p_f = 1$. Then the sets of probabilistic points-to relations at the merge point $join$ can be computed by meet operation.

$$G_{in}(l_{join}) = p_t \cdot G_{out}(l_{then}) + p_f \cdot G_{out}(l_{else})$$

Loop A loop can iterate an arbitrary number of times in a program due to the back-edge. Specifically, the back-edge of the loop is shown in Fig. 8(a) and the loop body B can be unfolded arbitrary times, as shown in Fig. 8(b). When handing the loop, two problems should be taken into account carefully. The first is how to estimate the

upper-bound and lower-bound of iteration number. The second is how to estimate the possibility for a given iteration number. The computation can be formulated as follows:

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

where α and β are the upper-bound and lower-bound of iteration number, F is the transfer function of loop body B.

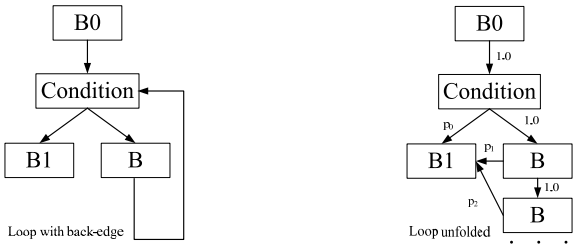


Fig. 8. Loop

In JPPA, only two situations are taken into account when dealing with loop:

1. The probability entering the loop with a fixed iteration number;
2. The probability not entering the loop.

The approximate of the iteration number is that after that number, the probabilistic points-to graph under the transfer function of the loop body is stable.

$$G_{i+1} = F(G_i)$$

The transfer function is the composition of the four kinds of basic functions. When the results of each iteration stable, the graph G satisfies the constrains with the following form:

1. $\Delta_v^G = \bar{o}$
2. $\Delta_v^G = \Delta_r^G$
3. $\Delta_v^G = \sum_{o \in \lceil \Delta_r^G \rceil} \Delta_r^G(o) \cdot \Delta_{o.f}^G$
4. $\Delta_{o.f}^G = \Delta_v^G(o) \cdot \Delta_r^G + (1 - \Delta_v^G(o)) \cdot \Delta_{o.f}^G, o \in \lceil \Delta_v^G \rceil$

The last formula can be simplified as follows:

$$\Delta_{o.f}^G = \Delta_r^G, o \in \lceil \Delta_v^G \rceil$$

For example, in Fig. 1, if the probability entering and not entering the loop is equal, then the distribution of `o1.Shape.area` and `o2.area` are changed after analyzing the

loop.

$$\begin{aligned}
\Delta_{o_1.Shape.area} &= \frac{1}{2}\bar{o_4} + \frac{1}{2}(\frac{1}{3}\bar{o_5} + \frac{1}{3}\bar{o_6} + \frac{1}{3}\bar{o_7}) \\
&= \frac{1}{2}\bar{o_4} + \frac{1}{6}\bar{o_5} + \frac{1}{6}\bar{o_6} + \frac{1}{6}\bar{o_7} \\
\Delta_{o_2.area} &= \frac{1}{2}\bar{o_3} + \frac{1}{2}(\frac{1}{3}\bar{o_5} + \frac{1}{3}\bar{o_6} + \frac{1}{3}\bar{o_7}) \\
&= \frac{1}{2}\bar{o_3} + \frac{1}{6}\bar{o_5} + \frac{1}{6}\bar{o_6} + \frac{1}{6}\bar{o_7}
\end{aligned}$$

The iteration is convergence under the four forms of the constrains mentioned above. In order to accelerate the speed of convergence, we set a small number ϵ. If the distance of the two graphs is smaller than ϵ, the two graphs can be seen as the same one.

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. Exception handling has effect on program analysis and testing [13].

For the program testing and debugging, probabilistic points-to information in the exception block is quite useful. 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** 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.

3.3 Interprocedural Analysis

The algorithm for interprocedural probabilistic points-to analysis is to perform the analysis crossing the boundary between methods. At each call site, points-to relationships are mapped from actual parameters to formal parameters by the algorithm, and the results are mapped back to the variables in the caller.

In JPPA, the interprocedural edges are carefully taken into account. At a method call site, partial PPG is propagated to the callee method. The partial PPG is a forest in which the root nodes of the trees are static fields and actual parameters. Algorithm 1 takes a root reference set and a PPG as input and computes all the objects and their fields which can be visited through the points-to relations. This partial PPG includes all the objects and their fields that may be effected by the called method. When passing the return value to the variable in the caller method, not only the return value but also all the points-to relation changes should be presented on the graph. Algorithm 2 describes the computation of PPG after method call. In the algorithm, the input PPG is a copy of the PPG before the method call. The input graph is updated using all the points-to relation changes caused by the analysis of called method.

Since Java programs heavily use libraries that contain many unused methods, a probabilistic interprocedural points-to analysis requires an approximation of the call graph to avoid analyzing unused code. This can be constructed in advance using a technique such as CHA [14], RTA [15] and VTA [16]. Another way is to keep track of all

```
input  : A probablistic points-to graph PPG and a reference set RefSet.
output: void
1.1  begin
1.2      if RefSet.isEmpty () then
1.3          return ;
1.4      end
1.5      foreach var ∈ RefSet do
1.6          if var.GetSupportSet (PPG).isEmpty () then
1.7              set.Add (var);
1.8              foreach object ∈var.GetSupportSet (PPG) do
1.9                  WorkSet.Add (object);
1.10             end
1.11         end
1.12     end
1.13     while WorkSet.isEmpty () do
1.14         oi ← WorkSet.GetElement ();
1.15         WorkSet.Remove (oi);
1.16         foreach field ∈ FieldSet do
1.17             if not (oi.field).GetSupportSet (PPG).isEmpty () then
1.18                 set.Add (oi.field);
1.19                 foreach oj ∈ (oi.field).GetSupportSet (PPG) do
1.20                     WorkSet.Add (oj);
1.21                 end
1.22             end
1.23         end
1.24     end
1.25     foreach var ∈ PPG.GetRefSet () do
1.26         if var ∉ set then
1.27             PPG.RemoveRef (var)
1.28         end
1.29     end
1.30 end
```

Algorithm 1: MakePoints-toClosure

```
input  : A probablistic points-to graph PPG at a call site and the corresponding
         probablistic points-to graph rPPG from method return.
output: void
2.1  begin
2.2      foreach var ∈ rPPG.GetRefSet () do
2.3          if var is a static field or a field access with the form "object.field" then
2.4              PPG.RemoveRef (var);
2.5              PPG.AddRef (var.Copy ())
2.6          end
2.7      end
2.8  end
```

Algorithm 2: UpdatePPG

reachable methods on-the-fly as the points-to set of the receiver variable is computed at each call site during the analysis. However, this approach gives somewhat higher precision and requires more iteration as the interprocedural edges are added to the ICFG.

The call graph as the input data of JPPA is used to construct the ICFG. Thus, the call graph can be built by CHA, RTA, or VTA algorithm. CFGs of methods can be created according to intraprocedural analysis and through each method call these CFGs can be combined with a whole program ICFG. At each call site, two kinds of auxiliary nodes are generated. One kind named entrance node deals with the parameters passing which records the map from actual parameters to formal parameters. And the other kind named return node handles the method return. All values returned by the method are assigned to an auxiliary variable included in the return node, which makes each method have a unique return variable.

Because JPPA is context-insensitive analysis which dose not distinguish the different contexts under which a method is invoked, it may share one callee method under different calling contexts. In this situation, the analysis should handle meet operation as follows:

$$G_{in}(l_m) = \sum_{cs_i \in CS} Prob(e_{cs_i}) \cdot G_{out}(PP_{cs_i}), e_{cs_i} = (l_{cs_i}, l_m).$$

where m means the callee method, $G_{in}(l_m)$ means the PPG before entering the method m, CS denotes the set of call sites with callee method m, $Prob(e_{cs_i})$ denotes the probability of method m called happens at cs_i, $G_{out}(PP_{cs_i})$ represents the PPG after parameter passing at cs_i.

Polymorphism Polymorphism is the capability of a method to do different things based on the receiver object. Polymorphism gives us the ultimate flexibility in extensibility. But it also brings to us some difficulties in static program analysis since it is hard to get the target method called in run-time at a call site. At a call site, the target method is contained in an approximation set of methods called.

$this$ as an implicit formal parameter should be mapped to the receiver object in the caller method as usual. In the case of polymorphism, $this$ should be carefully taken into account according to the type of the receiver object. In our framework, handling $this$ is a little different from handling usual formal parameters. Suppose the method $m()$ is declared in class C and at each call site cs_i ($i \in I$) method $m()$ may be called in the form $r_i.m()$. At each call site cs_i, there exits an object set

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

where $MatchedClass(C, m())$ is the class set which contains the class C and its subclasses not overriding method $m()$. The distribution of $this$ can be computed by the following equations

$$\begin{aligned} \Delta_{this} &= \sum_{i \in I} \sum_{o \in set_i} \frac{Prob(e_{cs_i}) \cdot \Delta_{r_i}(o)}{a} \overline{o} \\ a &= \sum_{i \in I} \sum_{o \in set_i} Prob(e_{cs_i}) \cdot \Delta_{r_i}(o) \\ e_{cs_i} &= (l_{cs_i}, l_m) \end{aligned}$$

where $Prob(e_{cs_i})$ denotes the probability of $m()$ being called at the call site cs_i.

For example, in Fig. 1, the distribution of `this` in method `Round.set` is $1.0\bar{o_1}$ not $0.25\bar{o_1} + 0.75\bar{o_1}$.

At a virtual call site, return value should also be considered. Suppose a virtual call site $s: \ v = r.m()$. According to the points-to set of r, the target methods with the same method signature can be deduced and represented in the form $m_j(), j \in J$. The distribution of variable v is calculated as follows

$$\Delta_v = \sum_{j \in J} Prob(e) \cdot \Delta_{return_{m_j}}, e = (l_s, l_{m_j})$$

where $Prob(e)$ denotes the probability of $m_j()$ being called at the call site s, $\Delta_{return_{m_j}}$ denotes the return value distribution of method $m_j()$.

For example, in Fig. 1, the distribution of variable `r` is $0.25\bar{o_5} + 0.75\bar{o_6}$.

Library Code So far as we know, user code and library code have intricate interconnections and one cannot perform points-to analysis on Java user code without considering libraries. Analyzing library code is also emphasized in this paper.

From the observations, we found that a lot of methods in library code called in the user code do not effect the points-to analysis. If these irrelevant methods are taken into account, more time and space are consumed while there is no precision improvement. For example, $System.out.println()$ covers 60156 lines of code in static single-assignment (SSA) form and 2229 methods in the library code. In these library code, 33680 object references and 4825 objects are involved. However, $System.out.println()$ does not change any points-to relations during the analysis. It does not lose any precision of the probabilistic points-to analysis and saving much time and space without considering these irrelevant code.

We also found that the small part of the library code tend to change the points-to relations. These classes are the java container classes under the package $java.util$. So, these class should be taken into account during the analysis.

Thus, a flexible measurement is taken. The coverage of the library code analyzed is controlled in JPPA. For different application program, the library code being analyzed can be configured in the file by package names.

3.4 Safety

Compared with the conversional context-insensitive and flow-sensitive points-to analysis, JPPA computes a safe result. A support set of the distribution Δ_v is the points-to set of the reference v, which can be represented as $Pt(v)$. The operation on the distribution is associated with the operation on the points-to set. Fig. 9 reveals different operation between probabilistic points-to analysis and conversional points-to analysis. Because of the flow-sensitive, when dealing with branch merging in the ICFG, the meet operator is $\bigcup$.

The conversional context-insensitive and flow-sensitive points-to analysis can be seen as the problem that computes the least fix-point over a finite lattice. After a finite

Statement	PPA	CPA
$v = new\ C$	$\Delta_v \leftarrow \bar{o}$	$Pt(v) \leftarrow \{o\}$
$v = r$	$\Delta_v \leftarrow \Delta_r$	$Pt(v) \leftarrow Pt(r)$
$v = r.f$	$\Delta_v \leftarrow \sum_{o \in \lceil \Delta_r \rceil} \Delta_r(o) \cdot \Delta_{o.f}$	$Pt(v) \leftarrow \bigcup_{o \in Pt(r)} Pt(o.f)$
$v.f = r$	$\Delta_{o.f} \leftarrow \Delta_v(o) \cdot \Delta_r + (1 - \Delta_v(o)) \cdot \Delta_{o.f}, o \in \lceil \Delta_v \rceil$	$Pt(o.f) \leftarrow Pt(r) \cup Pt(o.f), o \in Pt(v)$

Fig. 9. Comparison between PPA and TPA.

iteration, the points-to sets at each program points is stable. At this point, our analysis also calculates the same points-to sets as the conversional one.

In the previous sections, we discuss three structures which are sequence, branch and loop in intraprocedural analysis. It is easy to proof that our algorithm in these basic structure is convergence. For the interprocedural analysis, handling recursive procedures is important. From the call graph, the SCCs (strong connection component) are computed and the ICFG can be divided into small parts. According to their dependences, each part is analyzed by a certain order to reduce the iteration number. For a single SCC, the back-edges are recognized to break the loop. In order to ensure the safety and termination, a small ϵ is chosen to measure the distance between two probabilistic points-to graphs. In an SCC, the worklist algorithm is applied and the analysis terminates in line with the change distance of each PPG that is less than ϵ.

4 The JPPA Infrastructure

In this section an implementation of JPPA is described. Fig. 10 is a block diagram of JPPA infrastructure. The input of JPPA is Jimple [17, 18] code, a typed 3-address *intermediate representation* (IR) generated by Soot [19] from Java source code. The call graph by RTA algorithm is built by Soot and all reachable methods can be covered in order to avoid analyzing dead code. The intraprocedural CFG for each method is constructed. Then, JPPA begins by combining all CFGs to an ICFG according to the call graph. Each edge is annotated with a probability indicating the expected frequency of the edge during the execution. Next, all the references and objects are extracted from Jimple IR and each of them is given a global unique name and ID. Finally, the work list algorithm is applied to the nodes of ICFG and the points-to graph for each point of the program is computed.

To construct an ICFG, we first use Soot to trace the program from the main method and get all reachable methods and their invocation relations shown in the call graph. Then we combine all CFGs to build an ICFG. For each call site, we add two special nodes representing parameter passing and return value passing. We add an edge from the parameter passing node to the entry of the callee and an edge from the exit point of the callee to the call site. We also handle the exception mechanism by adding edges from each statement in the try block to the entry of the catch block. After the ICFG is constructed, we label each edge with their probabilities. For those edges lacking such information, we assume that all fan-in probabilities at all incoming branches, entitled as the reciprocal of the number of incoming branches as well, are of equal value.

To perform the analysis, all the reference variables and objects of the program are represented. The objects can be simply identified with object creation statement. The

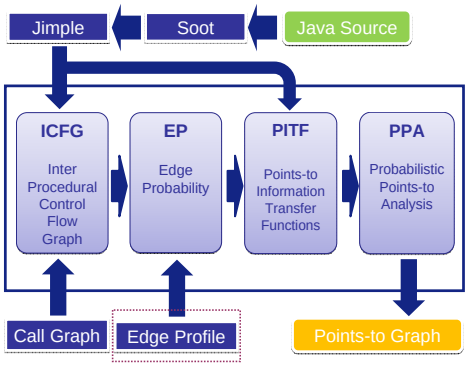


Fig. 10. The block diagram of JPPA infrastructure.

reference variables can be categorized into three groups: local variables, instance fields and static fields. When analyzing all the methods in ICFG, the local variables can be easily identified by its declaration in method body. However, each instance field can only be preliminarily extracted when encountering an object allocation site. In that case, fields in super classes are considered. Static fields can be extracted from all the classes of the program being analyzed.

Additionally, the inheritance of classes and the shadowing declarations of class fields are also taken into account. The class field access table is built to relate each class field visited to its static declaration according to the class hierarchy graph. For example, in Fig. 1 classes `Shape`, `Round` and `Square` are declared. The declaration of `area` in `Shape` is shadowed by the declaration of `area` in `Round`. The field access table of the program in Fig. 1 is illustrated in Fig. 11. The form `<classname: fieldclass fieldname>` denotes a field access, where `classname` means the static class of the reference variable, `fieldclass` means the field type and `fieldname` means the field name. For the statement `Shape p = new Round()`, the field access of `p.area` is `<Shape: Double area>`. When dealing with `Round p = new Round()`, the field access `p.area` is `<Round: Double area>`. From Fig. 11, `<Shape: Double area>` and `<Round: Double area>` can be distinguished, while `<Shape: Double area>` and `<Square: Double area>` are the same field access.

Field access	Field declaration
`<Shape: Double area>`	`Shape.area`
`<Round: Double area>`	`Round.area`
`<Square: Double area>`	`Shape.area`
`<Round: Double radius>`	`Round.radius`
`<Square: Double sideLength>`	`Square.sideLength`

Fig. 11. The field access table of the program in Fig. 1.

Once the ICFG and global names for all references and objects in the program are built, the work list algorithm can be applied. Each node is initialized with an empty points-to graph and added to the work list. Every time one node is retrieved from the work list and calculated the G_{in} and G_{out} points-to graph according to the transfer functions. And if G_{in} or G_{out} of the node is changed, compared to the previous one, all the successive nodes of it are added back into the work list. This procedure is repeated until the work list is empty.

5 Evaluating JPPA

We implemented our analysis using Soot 2.3.0 [19]. The analysis includes the Sun JDK 1.6.0_10 libraries. All experiments were performed on a machine with an AMD Sempron(tm) 1.80GHz CPU, and ran with 1G heap size (option -Xmx1024m). Table. 1 shows the set of benchmark programs, which includes the SIR suite [20], programs from the Ashes suite [21], programs from the DaCapo suite [22], and two other programs. The analysis begins with the main method and gets all the reachable methods in the call graph built by RTA algorithm. Table 1 illustrates the total number of statements (#Statements) in Jimple representation and the basic blocks (#Blocks) of these benchmark programs.

Table 1. Java Benchmarks.

Program	#Statements	#Blocks	Description
HashMap	20929	12307	A small program using HashMap in Java library.
ArrayList	150	27	A small program using ArrayList in Java library.
antlr	48343	20938	A parser generator and translator generator.(DaCapo)
xalan	20789	11507	An XSLT processor for transforming XML documents.(DaCapo)
luindex	21420	11897	A text indexing tool.(DaCapo)
hsqldb	21057	11617	An SQL relational database engine written in Java.(DaCapo)
toba-s	30850	14207	A tool translating Java class files into C source code.(Ashes)
Jtopas	32226	20858	A Java library for the common problem of parsing text data. (SIR)
JLex	32058	16111	A lexical analyzer generator for Java.
java_cup	37634	18049	A LALR parser generator written in Java.

5.1 Points-to Analysis Precision

The accuracy of a conventional points-to analysis algorithm is typically measured and compared by computing the average size of points-to sets. Table 2 illustrates the average sizes of the points-to sets for each benchmark studied, showing the max size of the points-to sets during the analysis and percentage of the points-to set with only one object. From Table 2, we can see that for each benchmark the average size of points-to sets is under 2 and the max size of the points-to sets is below 20. The average size of points-to sets is closer to 1, there are more useful information benefitting the traditional optimization. There are still large points-to sets in the benchmarks, for instance **tobas**

and **java_cup** which means it may have a lot of optimization chances for the speculative optimizations.

Table 2. JPPA Measurements.

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

5.2 Probabilistic Precision

We now measure the accuracy of the probabilities computed by JPPA by comparing the two probability points-to graphs G_s and G_d at some special program points which are before or after the basic blocks. G_s represents the probability points-to graphs generated statically. There are the following two variations of points-to analysis:

1. Probabilistic points-to analysis based on static probabilities (PPA). A probability is assigned to each incoming edge of ICFG which is equal for the edges to the same destination node, and the probabilistic points-to analysis algorithm is executed based on these edge probabilities;
2. Traditional points-to analysis (TPA). The probability of each points-to relation within a points-to set is assumed to be equal.

G_d represents the dynamic probability points-to graph calculated by the points-to profiler. In particular, we want to quantify the accuracy of the probability points-to graph G_s that are statically computed at the program points.

So far as we known, a software may contain a lot of functions, but only major functions are used usually by an user. If all the input data are taken into account equally, the analysis can not place emphasis on the program points covered by the major functions. So, the runtime probabilistic points-to graph data are gathered on the pathes executed as frequently as possible.

The distance of two probability points-to graphs can be calculated as follows:

$$distance = \frac{\sum \|G_s - G_d\|}{|program\_points|}$$

$$\|G_s - G_d\| = \frac{\sum\limits_{r \in Ref} \|\Delta_r^s - \Delta_r^d\|}{|Ref|}$$

where $program\_points$ means the the program points on the pathes we have studied. To compare the two distributions in a meaningful way, we compute the normalized Euclidean distance as defined by:

$$\|\Delta_r^s - \Delta_r^d\| = \frac{\sqrt{\sum_{o \in Obj} (\Delta_r^s(o) - \Delta_r^d(o))^2}}{\sqrt{2}}$$

The metric $distance$ summarizes the average error uniformly across all probability points-to graph at the program points on a scale that ranges from zero to one, where a zero means no discrepancy between dynamic and static graphs, and a one means there is always a contradiction at the program point.

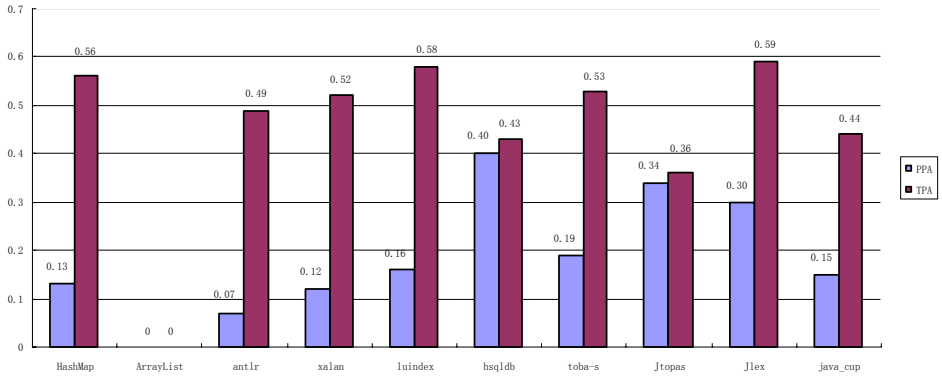


Fig. 12. Graph Distances.

Fig. 12 manifests the distance between the probability points-to graph created by PPA and TPA and by the runtime points-to profile. In most cases, PPA result is better than TPA result, and PPA average distance is 0.17. However, in the cases **hsqldb** and **Jtopas**, these two distances is very close. Through the investigation, we found that, the usage of the object array in these programs effects PPA. If the object array are heavily used in the source code, the precision of PPA drops to that of TPA.

5.3 Analysis Performance

Table 3 shows the running time and memory usage for the analysis and also reports the number of the references, static fields and the objects. After analyzing the result data, we detect that the distributions of the static fields directly effect the performance, since the static field distributions are propagated to all the nodes in the ICFG. If the distributions of the static fields have big support sets, the propagation consumes both time and space. Another important factor effecting the analysis performance is the usage of object array. Since the object array is handled by a conservative way that puts all its object elements into one points-to set. If the object array is used frequently in the program, both performance and precise of the analysis decline.

Table 3. Performance.

Program	#references	#static-fields	#objects	Time(Sec)	Memory(MB)
HashMap	4200	104	296	71.36	185.86
ArrayList	42	0	9	1.55	0.38
antlr	7835	136	910	53.33	313.33
xalan	4154	91	342	30.92	147.67
luindex	4318	91	363	31.62	162.85
hsqldb	4130	92	361	31.33	147.79
toba-s	5101	119	695	41.53	337.85
Jtopas	5896	107	440	139.03	319.38
JLex	5491	129	481	86.59	278.39
java_cup	6431	132	825	215.14	312.84

6 Related Work

In the past several years, points-to analysis has been an active research field. A very good overview of the current state of algorithms and metrics for points-to analysis are given by Hind [23].

Conventional points-to analysis. Context-sensitivity and flow-sensitivity are two major dimensions of pointer analysis precision. The most accurate algorithms are both context-sensitive and flow-sensitive (CSFS) [24–29]; however, these approaches are difficult to scale to large programs. Recently proposed idea of bootstrapping by [30] enables context-sensitive and flow-sensitive algorithms to scale. Context-insensitive and flow-insensitive (CIFI) algorithms [31, 32] have the best scalability on the large programs with overly conservative results. Equality-based analysis and subset-based analyses become the two wildly accepted analysis styles. In [33], Liang et al. test several variations of Java points-to analyses, including subset-based and equality-based variations.

How well the algorithms scale to large programs is an important issue. Trade-offs are made between efficiency and precision by various points-to analysis. One choice is context-sensitive and flow-insensitive (CSFI) analysis [3, 34, 4, 35]. The other is context-insensitive and flow-sensitive (CIFS) analysis [5, 6]. However, all these conventional points-to analyses can not give the probability for the may-be points-to relation.

Probabilistic pointer analysis for C. With the proposition of the speculative optimizations, the probability theory is introduced into the conventional program analysis. In earlier work, Ramalingam [12] proposes a generic data flow frequency analysis framework that can compute the probability a fact holds true at every control flow node using the edge frequencies propagation. It provides a formulation that can be used for probabilistic points-to analysis based on data flow analysis framework.

Chen et al. [36, 37] develop CSFS probabilistic point-to analysis algorithm. Their 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. Their interprocedural approach is based on Emami's algorithm [24] and therefore loses some scalability. Their experimental results demonstrates that their technique can estimate the probabilities of points-to relations in benchmark programs with reasonable accuracy although they did not disambiguate heap and array elements. JPPA

is also based on an iterative data flow analysis framework but with context-insensitive analysis. The conception of the discrete probability distribution are introduced. In every control flow node, the state of the program is a probabilistic points-to graph which can be represented as a set of the distributions and every transfer function operates on these distributions.

Silva and Steffan [38] propose 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. Through the experiments, they demonstrate that their analysis can provide accurate probabilities. However, in their framework, they use a one-level unsafe transformation to model load and store assignment, which ignores the alias between the shadow variables. JPPA avoids this unsafety for that the transfer functions of instance field read and write propagate the distribution without using the shadow variables.

7 Conclusions

In this paper we have presented JPPA, a probabilistic points-to algorithm for Java that is context-insensitive and flow-sensitive. JPPA predicts the likelihood of points-to relations without depending on expensive runtime profiles. In order to ensure the safety of the result, JPPA takes Java library code into account. JPPA has a flexible mechanism which enables adjusting the probability of each edge during the analysis. We have compared JPPA with the conventional context-insensitive and flow-sensitive points-to analysis. The experimental results manifest that the probabilistic points-to graph computed by JPPA is closer to the condition that software is used in practice. In addition, JPPA can produce accurate probabilities for each points-to relation which uses static heuristics rules.

In the future research, we would like to improve the convergence of our analysis algorithm by reasonable partition of the ICFG. In order to promote the precise of the object array analysis, we would also like to introduce abstract interpretation and domain theory to analyze the index of the array. Finally, we would like to extend JPPA by copying the method nodes in the call graph for a context-sensitive analysis.

Acknowledgements

We would like to thank Hongshen Zhang, Yu Kuai and Cheng Zhang for their discussion on this work.

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. (2001) 260–278
2. Hind, M., Pioli, A.: Which pointer analysis should I use? In: International Symposium on Software Testing and Analysis. (2000) 113–123
3. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for Java. ACM Transactions on Software Engineering and Methodology (TOSEM) **14** (2002) 1–41

4. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation. (2004) 131–144
5. 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, ACM Press (1993) 232–245
6. Hardekopf, B., Lin, C.: Semi-sparse flow-sensitive pointer analysis. In: POPL '09: The 36th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages. (2009)
7. Dai, X., Zhai, A., chung Hsu, W., chung Yew, P.: A general compiler framework for speculative optimizations using data speculative code motion. In: CGO. (2005) 280–290
8. Lin, J., Chen, T., Hsu, W.C., Yew, P.C.: Speculative register promotion using advanced load address table (alat). Code Generation and Optimization, IEEE/ACM International Symposium on **0** (2003) 125
9. Lin, J., Chen, T., Hsu, W.C., Yew, P.C., Ju, R.D.C., Ngai, T.F., Chan, S.: A compiler framework for speculative analysis and optimizations. In: PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation. (2003) 289–299
10. Kazi, I.H., Lilja, D.J.: JavaSpMT: A speculative thread pipelining parallelization model for Java programs. In: Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS, IEEE (2000) 559–564
11. Pickett, C.J.F.: Software speculative multithreading for Java. In: OOPSLA'07 Companion: Companion to the Proceedings of the 22nd ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications, New York, NY, USA, ACM (2007) 929–930
12. Ramalingam, G.: Data flow frequency analysis. In: PLDI '96:Proceedings of the 1996 conference on Programming language design and implementation. (1996) 267–277
13. Sinha, S., Harrold, M.J.: Analysis and testing of programs with exception handling constructs. IEEE Transactions on Software Engineering **26**(9) (2000) 849–871
14. Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static class hierarchy analysis. In: ECOOP95: Object-Oriented Programming 9th European Conference. (1995) 77–101
15. 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 (OOPSLA '96). (1996) 324–341
16. Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E., Godin, C.: Practical virtual method call resolution for Java. In: Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '00), ACM Press (2000) 264–280
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. 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** (2004) 405–435
21. Ashes Suite Collection: http://www.sable.mcgill.ca/software
22. 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: OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications, New York, NY, USA, ACM Press (October 2006) 169–190
23. Hind, M.: Pointer analysis: Haven't we solved this problem yet? In: 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01), Snowbird, UT (2001)
24. Emami, M., Ghiya, R., Hendren, L.J.: Context-sensitive interprocedural points-to analysis in the presence of function pointers. In: PLDI '94: Proceedings of the 1994 conference on Programming language design and implementation. (1994) 242–256
25. Landi, W., Ryder, B.G., Zhang, S.: Interprocedural modification side effect analysis with pointer aliasing. In: PLDI '93: Proceedings of the 1993 Conference on Programming Language Design and Implementation, ACM Press (1993) 56–67
26. Landi, W., Ryder, B.G.: A safe approximate algorithm for interprocedural pointer aliasing. Proceedings of the Conference on Programming Language Design and Implementation **2622** (1992) 235–248
27. Whaley, J., Rinard, M.: Compositional pointer and escape analysis for Java programs. In: Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications, ACM Press (1999) 187–206
28. Wilson, R., Lam, M.S.: Efficient context-sensitive pointer analysis for C programs. In: PLDI '95: Proceedings of the 1995 conference on Programming language design and implementation, ACM Press (1995) 1–12
29. Whaley, J., Lam, M.S.: An efficient inclusion-based points-to analysis for strictly-typed languages. In: Proceedings of the 9th International Static Analysis Symposium. (September 2002)
30. Kahlon, V.: Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation, ACM Press (2008) 249–259
31. Steensgaard, B.: Points-to analysis in almost linear time. In: Symposium on Principles of Programming Languages. (1996) 32–41
32. Andersen, L.: Program analysis and specialization for the C programming language. DIKU report 94-19, University of Copenhagen (1994)
33. Liang, D., Pennings, M., Harrold, M.J.: Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. (2001) 73–79
34. Zhu, J., Calman, S.: Symbolic pointer analysis revisited. In: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation, ACM Press (2004) 145–157
35. Lhoták, O., Hendren, L.J.: Context-sensitive points-to analysis: Is it worth it? In: International Conference on Compiler Construction (CC). (2006) 47–64
36. Chen, P.S., Hung, M.Y., Hwang, Y.S., Ju, R.D.C., Lee, J.K.: Compiler support for speculative multithreading architecture with probabilistic points-to analysis. In: Proceedings of the ACM SIGPLAN symposium on principles and practice of parallel programming (PPoPP 2003), ACM Press (2003) 25–36
37. 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)** (2004) 893–907
38. Silva, J.D., Steffan, J.G.: A probabilistic pointer analysis for speculative optimizations. In: ASPLOS '06: Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems. (2006) 416–425