# Scaling Context Sensitive Points-to Analysis by Geometric Encoding

Xiao Xiao, The Hong Kong University of Science and Technology Charles Zhang, The Hong Kong University of Science and Technology

Context sensitive points-to analysis, while significantly benefiting many static analysis techniques, is known to be difficult to scale to large programs. We have designed the geometric encoding, a novel technique to effectively capture the redundancy in representing a large number of contexts without using BDD. Compared to the state-of-the-art data compression techniques, such as BDD and EPA, geometric encoding is also capable of evaluating contexts of points-to constraints *in the compressed form directly*, but incurring much less space and time requirements. Our experiments show that, to the best of our knowledge, geometric encoding is the first technique to perform the *context sensitive* analysis on large Java benchmarks with JDK 1.6 libraries. When evaluated on smaller subjects, our technique is 7.1X and 81.9X faster than the worklist based 1-object sensitive analysis in Paddle, with the precision comparable or better when compared to the 1-object-sensitive analysis. Our reference implementation is now a part of the official distribution of Soot, a widely used framework for analyzing Java programs.

Categories and Subject Descriptors: F.3.2 [Semantics of Programming Languages]: Program analysis

General Terms: Algorithms, Languages, Performance

Additional Key Words and Phrases: Geometric Encoding, Points-to, Efficient, Scalable

#### 1. INTRODUCTION

The static points-to analysis determines, given a pointer p, the set of abstract memory locations that p may point to. The recent study [Fähndrich et al. 2000; Cheng and Hwu 2000; Liang and Harrold 2001; Whaley and Lam 2004; Zhu and Calman 2004; Milanova et al. 2005; Sridharan and Bodík 2006; Lattner et al. 2007; Xu and Rountev 2008; Yu et al. 2010] of points-to analysis focus on improving the precision of points-to results by leveraging the context sensitivity. Context is a static abstraction to distinguish the different runtime invocations of the same function, typically represented by call-string [Lhoták 2006], a sequence of callsites leading to the execution of a function. The context-sensitive points-to analysis is commonly reasoned under the k-CFA  $(k \ge 0)$  analysis framework [Shivers 1991], where k is the context depth that indicates the length of the path ascending from any function on the call graph. Our focus in this paper is to design a practical points-to analysis abstracted by the *full context* sensitivity for both the pointer and the heap variables [Nystrom et al. 2004], where the term *full* means the k is the length of the longest path in the call graph with the strongly-connected components (SCCs) contracted. We make this decision because the full context sensitivity is the most precise case in theory and, there is a simple numerical representation for full context sensitivity, which can significantly simplify the design and implementation of the points-to analysis.

© YYYY ACM 1539-9087/YYYY/01-ARTA \$10.00

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

This research is supported by RGC GRF grants 622208 and 622909. The preliminary version of this paper titled "Geometric Encoding: Forging high performance context sensitive points-to analysis for Java" is published at ISSTA 2011.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

# 1.1. Challenges

We are faced with three major challenges of scaling the context sensitive analysis to full context sensitivity with a large context depth k:

**1. The number of contexts is exponential to** k. For example, our smallest benchmark jflex has  $3.9 \times 10^9$  contexts, and another benchmark pmd has more than  $2^{63}$  contexts. The sheer quantity of contexts demands very compact representations in the analysis.

**2.** The number of pointer constraints also grows exponentially with k, since they need to be duplicated under every context. In order to be practical, the context sensitive constraints require high throughput evaluation algorithms.

**3. Recursive calls.** The call graph may have strongly-connected components (SCC) due to self or transitively induced recursions. Since a SCC produces a unbound number of contexts for its member elements, it is usually contracted to a single node, causing its members to be treated context insensitively. Because the Java programs always contain large SCCs (Section 6)<sup>1</sup>, we need a context sensitive treatment for SCCs to achieve good precision.

We now discuss the limitations of the recent progress in light of tackling these scalability challenges and comparatively sketch our solution.

#### 1.2. Prior Work

Sharir and Pnueli proposed the call-string and functional approaches in their seminal paper [Micha and Pnueli 1981], which laid the foundation for the context sensitive program analysis. The *call-string* approach analyzes the program as a whole, abstracts every variable and heap location with a static call path, in order to distinguish the different runtime instances of the syntactically identical variables and heap memory locations. Many pioneer algorithms [Landi and Ryder 1992; Emami et al. 1994; Wilson and Lam 1995; Hind et al. 1999] adopt the call-string approach. The *functional* or *modular* approach summarizes the memory effects for each function independent of any calling context in advance, then inlines the function summaries at all the callsites to obtain the collective points-to results. The early scalable approaches [Chatterjee et al. 1999; Fähndrich et al. 2000; Cheng and Hwu 2000; Liang and Harrold 2001] fall in this category. However, none of these literature reports excellent scalability because the points-to or alias information is not efficiently represented and propagated.

Whaley *et.al.* [Whaley and Lam 2004] have presented the first practical solution that uses the *cloning* based treatment to achieve the full context sensitivity on very large realistic Java programs. They have successfully used BDD to compress the enormous number of context sensitive points-to tuples and to evaluate the points-to constraints in the compressed form. Acknowledged by the authors themselves [Whaley 2007], the memory efficiency of BDD is heavily influenced by the insertion order of the variables, requiring lots of tuning work in practice. More importantly, recent studies show that BDD is not powerful enough to encode multiple context-sensitive variables simultaneously, as in the case of heap cloning [Lhoták and Hendren 2008]. Besides, Hardekopf *et.al.* [Hardekopf and Lin 2007a] reported that the use of BDD in Anderson's algorithm for C is around 2x slower than the sparse bitmap encoding. Bravenboer *et.al.* [Bravenboer and Smaragdakis 2009a] even achieved 15x speedup on average with the Doop engine, compared to the BDD based context sensitive analyses in Paddle [Lhoták 2006]. Based on these results, BDD is limited in achieving our design objective of offering a practical analysis for both pointer and heap sensitivity.

 $<sup>^{1}</sup>$ C programs also have large SCCs [Cheng and Hwu 2000], although they are less and smaller than those in the java programs.

Some researchers also have made significant progress in scaling the points-to analysis using the modular approach [Liang and Harrold 2001; Lattner et al. 2007; Xu and Rountev 2008]. The flexibility of the modular design requires an additional effort for the parameters instantiation and the computation of the escaped information, as compared to the whole program analysis. Lattner et.al.'s work [Lattner et al. 2007] is quite scalable due to the use of Steensgard's unification method for the pointer assignments. Xu et.al.'s work [Xu and Rountev 2008] employ the same unification method, e.g. for the pointers appearing at the call sites. Additionally, Xu et.al. have designed a compression technique for eliminating redundancy in the traditional points-to representation, aimed at minimizing the time and memory consumption. They find that many points-to tuples share the same calling context prefixes and, by removing them, two points-to tuples become identical<sup>2</sup>. Xu et.al. call the shared prefix the equivalent context and design the context merging technique, EPA, to leverage the observation. Although merging the equivalent contexts is an insightful idea, the compression opportunity is not fully exploited (Section 3.4). Moreover, despite that the performance of EPA is significantly better than the BDD based Paddle points-to solver [Lhoták 2006], its absolute running time is still unsatisfactory for the practical use. As an example, the middle sized programs (approx. 5000 methods) analyzed with JDK 1.3 use 300 to 800 seconds execution time.

#### 1.3. Our Contribution

To address the shortcomings of both the BDD and the functional based algorithms, we have designed and evaluated a *call-string* based context sensitive points-to analysis *without using BDD*. More precisely, our algorithm is a context sensitive, flow insensitive, and field sensitive points-to analysis that makes use of three novel techniques:

**1.** We develop a new context encoding scheme called the *Geometric Encoding*, which concisely represents the points-to and the pointer assignment relations as regular geometric figures. Compared to EPA, the geometric encoding is simpler to implement and superior in terms of compression capability (See section 3.4). Compared to BDD, our encoding is much more time efficient and more flexible for performing the time and the precision trade-off;

**2.** We develop a preprocessing technique called *constraints distillation*. In Java, most of the constraints extracted for points-to analysis come from the Java library. However, not all of the library code affects the points-to calculation of the user's code. By wiping out the inconsequential library code prior to the main analysis procedure, we improve the performance without any precision loss;

**3.** The third technique is a *1*-CFA model for handling the recursive calls. As shown by Lhoták and Hendren [Lhoták and Hendren 2008], the major source of precision loss in the full context sensitive analysis is the imprecise handling of recursive calls, due to the fact that the Java programs commonly introduce big SCCs (Section 6). We overcome the difficulty by providing a novel *1*-CFA model built on top of the existing k-CFA abstraction, which effectively avoids the precision and performance degradation incurred by the large SCCs.

With these techniques, our algorithm can analyze large programs (more than 26,000 methods) with JDK 1.6, which is a major achievement in the context sensitive points-to analysis for Java. The reference implementation of our algorithm has been accepted by Soot [Vallee-Rai et al. 2000] since version 2.5<sup>-3</sup>, a popular framework for experiment-

<sup>&</sup>lt;sup>2</sup>For example,  $(p, \zeta_1, o, \zeta_2)$  denotes that the pointer p under context  $\zeta_1$  points to the object o allocated under context  $\zeta_2$ .

<sup>&</sup>lt;sup>3</sup>http://www.sable.mcgill.ca/soot/

ing Java code analysis in academia. Interested readers are encourage to download and use our algorithm in your own work.

## 1.4. Organization

The paper is organized as follows. We first present the background knowledge in Section 2 to the readers who are unfamiliar to the context sensitive points-to analysis. Section 3 provides the details of geometric encoding, including both the inference rules and an improved context abstraction for SCC. Section 4 introduces the constraints distillation technique and many other technical optimizations, and then induces our main points-to algorithm. Section 5 describes the important implementation choices. Finally, we exhibit our experimental results in Section 6, discuss the related work in Section 7 and conclude our paper in Section 8.

#### 2. PRELIMINARIES

Before moving to the full detail of our algorithm, we first introduce the theory of context abstraction, rigorously define the full context sensitivity and prove its soundness. Then, we introduce a simple points-to analysis that achieves the context sensitivity by method cloning, the idea of which is first appeared in Emami *et.al*'s paper [Emami et al. 1994] and developed by Zhu *et.al* [Zhu and Calman 2004] and Whaley *et.al* [Whaley and Lam 2004]. Knowing how the simple algorithm works is helpful to digest our sophisticated algorithm presented in the next section.

# 2.1. Context Abstraction

Briefly speaking, our points-to algorithm is a context sensitive version of the Anderson's analysis [Andersen 1994]. Therefore, to demonstrate the usefulness and correctness of our algorithm, we should answer the questions: What is context sensitive? Why do we need context sensitivity? Why is it a sound program abstraction? These three questions are the theme of this section.

In the vocabulary of abstract interpretation, the static analysis is an algorithm that computes the solution of a set of equations over a finite abstract domain. The equations express the semantics of the concerned constructs in a program, and the abstract domain defines the symbols used in the equations and spans the solution space. *Context* is a way to create a precise abstract domain of the points-to analysis. Since points-to analysis only concerns the connections between the memory locations, its abstract domain contains only the local, global and heap variables <sup>4</sup>. Since any global variable has fixed memory address at runtime, it has only one runtime instance across the program lifetime. Therefore, we only need to create one symbolic variable for each global variable in the abstract domain. However, the number of local variables are unbounded because the lexically identical local variable creates a new running instance each time its enclosing function executed. The situation is the same for heap variables. Hence, we need a *context function* to map the elements in an infinite domain to a finite domain, which is formally defined as follows.

**DEFINITION 1.** A runtime function instance is an invocation of that function.

**DEFINITION 2.** A **Context function** is a function  $f : \mathbb{R} \to \mathbb{S}$  that maps all the runtime function instances in the infinite domain  $\mathbb{R}$  to a finite domain  $\mathbb{S}$ .

With the context function, context can be defined formally:

DEFINITION 3. A **Context** is an element in the domain S.

ACM Transactions on Embedded Computing Systems, Vol. V, No. N, Article A, Publication date: January YYYY.

A:4

<sup>&</sup>lt;sup>4</sup>We refer to the dynamic allocated memories (*e.g.* malloc in C, new in C++ and Java) as heap variables.

The call-string approach maps the runtime function instances to a set of callsites strings (call-string). Here, the *callsite* only refers to the location of a function call in the program or, more precisely, an edge in the call graph. The call-string approach, in fact, defines a family of functions termed k-CFA, because we can limit the length of call-string to its last k callsites (k-call-string) [Shivers 1991]. The k-call-string can be constructed inductively. For example, a k-call-string for function **foo** is  $c_1c_2...c_k$  and there is a callsite c' in **foo** invoking **bar**, the induced k-call-string for **bar** can be constructed as  $c_2c_3...c_kc'$ . To prove the soundness of the k-CFA scheme, we first define an auxiliary function *callpath*.

DEFINITION 4. callpath is a function that maps the runtime instances of X to their runtime call paths, and a runtime call path is an invocation chain of that runtime instance.

Next, we prove the soundness of k-CFA scheme in Theorem 2.1.

#### THEOREM 2.1. *k*-CFA is a context function.

PROOF. According to Definition 2, it is sufficient to prove that the k-CFA context function is a function map and its codomain is finite.

#### 1. It produces finite number of call-strings (the codomain $\mathbb{S}$ is finite).

Suppose we have *n* functions and *m* callsites, then for any function *X*, the number of *k*-call-string of *X*, denoted by  $CS_k(f)$ , is at most  $m^k$ . Therefore, the total number of call-strings is bounded by  $O(nm^k)$ .

### 2. It maps every element in the infinite domain to the finite domain.

For every function X, we create  $CS_k(f)$  static instances. Every runtime instance of X is mapped to one of its static instance as follows. First, we use *callpath* to map the runtime instances of X to their runtime call paths. Second, since all possible runtime call paths are included in the static call graph and every path on the static call graph is mapped to one and only one k-call-string, hence in turn, every instance of X is mapped to a k-call-string. This proves that the k-CFA mapping is a function.  $\Box$ 

Particularly, the case k = 0 is called *context insensitive* context function, *i.e.*, every function only has one context. Therefore, the abstract domain for a context insensitive analysis can be built by simply collecting all the variables in the program. Because of its simplicity and efficiency, for many approaches [Whaley and Lam 2004; Xu and Rountev 2008; Sridharan and Bodík 2006; Yan et al. 2011] including ours in this paper, performing a context insensitive analysis is the first step to collecting the program information to assist the subsequent analysis.

In contrast, the *full context sensitivity* function does not restrict the length of callstrings. However, this results in infinite number of call-strings due to the recursive calls. To deal with the infinity, we should first identify the strong connected components (SCC) in the call graph to build an acyclic *hierarchical call graph*, where each hyper node represents an SCC and maintains the full call relationships in that SCC. We distinguish the call edges inside <sup>5</sup> and outside the SCCs and call them *cyclic calls* and *acyclic calls* respectively. Based on the hierarchical call graph, the full context function *fc* is defined as follows.

**DEFINITION 5.** The full context function fc consists of three parts: 1. Domain  $\mathbb{R}$ : the runtime function instances;

<sup>&</sup>lt;sup>5</sup>Inside means two ends of that edge are both in the same SCC.

ACM Transactions on Embedded Computing Systems, Vol. V, No. N, Article A, Publication date: January YYYY.

2. Codomain S: the set of call paths from the **main** function to all the functions X on the hierarchical call graph;

3. Mapping rule fc(a) = p: let p' = callpath(a), then p is the call-string with all the cyclic calls removed from p'.

Theorem 2.2 proves the soundness of the context function *fc*.

THEOREM 2.2. The full context function fc is sound.

PROOF. The codomain S of fc is finite because the hierarchical call graph is an acyclic graph and the number of paths for an acyclic graph is finite. According to the definition of p, p is unique and it is exactly the call-string for p' on the hierarchical call graph. Since S contains all the call-strings for the hierarchical call graph,  $p \in S$ . Therefore, fc maps every element in  $\mathbb{R}$  to only one element in S, *i.e.*, fc is sound.  $\Box$ 

## 2.2. A Cloning based Points-to Algorithm

In this section, we construct a simple points-to algorithm on top of the context sensitive abstract domain built by fc, the pseudo-code of which is given in Algorithm 1. This algorithm is *Cloning-based* and it has three steps. First, it creates the cloned call graph (Lines 2–14) for the input hierarchical call graph. Second, it builds the abstract domain (Lines 15–24) and generates the constraints (Lines 25–37). Third, an Anderson's analysis [Andersen 1994] is performed to solve the constraints. The first two preparing steps are explained in detail as follows.

**1. Building cloned call graph**. To build the cloned call graph, we first compute a context insensitive call graph via Anderson's analysis. Then, we contract the cycles and build the hierarchical call graph  $G_h$ , which is the input for Algorithm 1. To process the real Java programs, we add an artificial function **SuperMain** to  $G_h$  as the entry point. **SuperMain** calls all possible entry points for a Java program, including the **main** function, the static class initializers, and the functions called by JVM at startup. In this way, these run-only-once functions are analyzed without contexts, which is similar to the enhancement used for Doop [Bravenboer and Smaragdakis 2009b]. The modified  $G_h$  is exemplified in Figure 2. Next, we traverse the graph  $G_h$ , create a clone for each function X by using copy-and-paste to generate a new function **f** for each of its call-string and link each callsite to its clone **f**, as exemplified in Figure 2(b). When we generate a clone of an SCC node (Line 9), all the cyclic calls and the functions belonging to that SCC are duplicated. In this way, the functions in the SCCs are treated context insensitively. It is trivial to know that this cloned call graph contains all the call paths in the codomain of  $f_c$ .

**2.** Building abstract domain and constraints. We first put all the global variables into the abstract domain *AD*. Then, at Line 16, we visit all the cloned functions and put all the local variables into *AD*. Meanwhile, if a heap allocation site is found, we create a distinct symbol to represent this allocation site (Line 21) and put it into *AD*. This treatment is called *heap cloning* or *heap sensitivity*.

Based on the abstract domain, we scan the source code and transform the pointer related statements into constraints. A *constraint* is a transfer function that describes effect to the abstract program state when executing this statement. For example, each time executing the pointer assignment p = q merges the points-to result of q into p and we formulate this semantics as  $p \supseteq q$ . Other statements are processed in the standard way [Rountev et al. 2001].

Algorithm 1 is essentially the cloning based context sensitive analysis [Whaley and Lam 2004; Zhu and Calman 2004] that straightforwardly integrates the context function fc into the Anderson's analysis, thus following Theorem 2.2, its correctness is immediate. However, it is impractical because its size of the cloned call graph and the

**Algorithm 1** The cloning based points-to analysis.

```
1: procedure CLONINGANALYSIS(G<sub>h</sub>: Hierarchical Call Graph)
                                                        \triangleright Part 1: Create the cloned call graph G_c
2:
        for all function X do
                                           \triangleright csize(X) records the number of call-strings for X
3:
            csize(X) = 0
        end for
4:
5:
        csize[main] = 1
        for all node X in G_h in topological order do
                                                                           \triangleright Node X may be an SCC
6.
7:
            for all acyclic call X \to Y do
                csize(Y) = csize(Y) + csize(X)
8:
9:
                Create csize(X) copies of Y named Y_1 \cdots Y_{csize(X)}
                for i \leftarrow 1 to csize(X) do
10:
                    We duplicate the acyclic call X_i \rightarrow Y_i
11:
                end for
12:
            end for
13:
14:
        end for
                                                        ▷ Part 2: Create the abstract domain AD
        AD \leftarrow All the global variables
15:
        for all function X in the cloned call graph G_c do
16:
            for all local variable l in X do
17:
                AD = AD \cup l
18:
            end for
19:
            for all heap allocation site p = \mathbf{new} XXXX in Xx do
20:
                Create a new symbol o for the expression new XXXX
21:
22:
                AD = AD \cup o
23:
            end for
        end for
24:
                                                             \triangleright Part 3: Generate the constraints \mathbb{C}
        \mathbb{C} \leftarrow \emptyset
25:
        for all statements in the cloned call graph G_c do
26:
27:
            if the statement is p = \mathbf{new} XXXX then
28:
                We lookup the symbol o for the expression new XXXX
                \mathbb{C} \leftarrow \mathbb{C} \cup "p \supseteq \{o\}''
29:
            else if the statement is p = q then
30:
                \mathbb{C} \leftarrow \mathbb{C} \cup "p \supseteq q''
31:
32:
            else if the statement is p f = q then
                \mathbb{C} \leftarrow \mathbb{C} \cup "p.f \supseteq q''
33:
            else if the statement is q = p.f then
34:
35:
                \mathbb{C} \leftarrow \mathbb{C} \cup "q \supseteq p.f''
            end if
36:
37:
        end for
                                                                     \triangleright Part 4: Solve the constraints
38:
        Apply the Anderson's analysis to \mathbb{C}
39: end procedure
```

abstract domain AD, and the number of the constraints  $\mathbb{C}$  are all exponential to the length of the longest path in the call graph. To make it useful, Whaley *et.al* [Whaley and Lam 2004] and Zhu [Zhu and Calman 2004] leverage the BDD data structure to compress the abstract domain and conduct the analysis in a compressed form. Their methodology successfully lowers the processing time down to a practical level. How-

ever, they compromise that the heap variables are handled context insensitively <sup>6</sup>, thus in turn, damage the analysis precision. We design a new data compression scheme called the *geometric encoding* to deal with the space and time explosion problem. This new approach is specially designed for heap sensitivity and it allows us to build a more precise full context function *fc*. In the next section, we will replace the components Part 1–3 in Algorithm 1 with new ones that can compress the cloned graph, abstract domain AD, and the constraints set  $\mathbb{C}$  to linear size. Then, we will redesign Part 4 that can compute the fixed point solution on the compact form.

## 3. GEOMETRIC ENCODING

Geometric encoding is a technique that compresses the abstract domain and rewrites the constraints semantics based on the encoded abstract domain. It also offers an inference system to evaluate the constraints in the encoded form, hence, it lays the foundation for computing the collecting semantics in a compact way.

In this section, we first give an example to informally illustrate our high level idea. The content of the example is rich enough to cover all the aspects of using geometric encoding for points-to analysis. In the rest of the subsections, we elaborate our observation and give a formal definition and proof for the geometric encoding.

## 3.1. Example and Insight

To help us present our technique, we symbolize the well known terms first.

 $p \mapsto o$ . : p points to o;

 $p \rightsquigarrow q$ . : *p* assigns to *q*, *i.e.* the abbreviation of the assignment q = p;

pts(p). : The points-to set of pointer p.

We illustrate our main insight using the sample code in Figure 1, which is a linked list exercise. The merge function is shown in the single static assignment form (SSA) [Cytron et al. 1991], because we can enjoy the limited flow sensitivity in this way [Hasti and Horwitz 1998]. The major challenge for analyzing the program is that all the objects are created at the two allocation sites in the Lines 12 and 35. Thus, we need the heap sensitivity to distinguish the objects created under different invocation paths of the allocations. Suppose we label the contexts of each function by the integers 1, 2

...N, where N is the total number of acyclic paths from SuperMain to that function on the hierarchical call graph (Figure 2), our aim is to conclude that every version <sup>7</sup> of the instance field  $o^{35}$ .next points to all versions of  $o^{12}$ .

To achieve this, Algorithm 1 evaluates the statements *Node*  $t1 = \mathbf{new} Node()$  (Line 12),  $p_2 = \phi(p_1, p_3)$  (Line 14),  $p_2.next = gList$  (Line 19), and gList = t1 (Line 20) many times under different contexts. However, with the following two observations, this computation can be dramatically simplified. First, the statement *Node*  $t1 = \mathbf{new} Node()$  is evaluated under 3 contexts *uniformly* and *independently*, because  $o^{12}$  **under the**  $i^{th}$  **context only assigns to** t1 **under the**  $i^{th}$  **context**. The evaluation result is  $t1_1 \mapsto o_1^{12}$ ,  $t1_2 \mapsto o_2^{12}$  and  $t1_3 \mapsto o_3^{12}$ . To help us study the pattern, we formalize the representation as  $t1_y \mapsto o_x^{12}$ ,  $\forall x \in [1, 4)$ , y = x. Note that, considering only the subscripts x and y, they form a *one-to-one mapping* that can be geometrically interpreted as a diagonal line segment y = x,  $x \in [1, 4)$ . The evaluation of  $p_2 = \phi(p_1, p_3)$  (Line 14) is in the similar way. For illustration purpose, we only explain the assignment  $p_2 = p_1$  for the  $\phi$ -function and the assignment  $p_2 = p_3$  is evaluated in the same way. Since  $p_1$  possesses the values

ACM Transactions on Embedded Computing Systems, Vol. V, No. N, Article A, Publication date: January YYYY.

A:8

 $<sup>^{6}</sup>$ At Line 21, we only create once for every new expression in the original program, rather than creating once for each occurrence in the cloned program.

<sup>&</sup>lt;sup>7</sup>The pointer p under different contexts are spoken as different versions of p.

1 cla	ass Main {	24 class Node {
2	Node gList = null;	25 int id;
3	public static void main(String[] args) {	26 public Node next;
4	Node list1 = prepare( 121 );	27 }
5	Node list2 = prepare( 34 );	28
6	Node list3 = <i>prepare</i> ( 56 );	29 class List {
7	<i>merge</i> ( list1 );	30 public static List prepare( int n ) {
8	Node q2 = <i>merge</i> ( list2 );	31 List t2 = List.createList( n );
9	<i>merge</i> ( list3 );	32 return t2;
10	}	33 }
11	public static Node merge(Node p1) {	34 public static Node createList( int n ) {
12	Node t1 = new Node(); // 0 <sup>12</sup>	35 Node t3 = new Node(); // 0 <sup>35</sup>
13	t1.next = p <sub>1</sub> ;	36 initialize( t3 );
	loop_start:	36 t3.next = setNext( n );
14	$p_2 = \Phi (p_1, p_3);$	37 return t3;
15	if ( p <sub>2</sub> .next != null ) {	38 }
16	$p_3 = p_2.next;$	39 public static Node setNext( int n ) {
17	goto loop_start;	40 Node nxt = <i>createList</i> ( n – 1 );
18	}	41 return nxt;
19	p <sub>2</sub> .next = gList;	42 }
20	gList = t1;	43 public static void setID( Node t4 ) {
21	return t1.next;	44 t4.id = -1;
22	}	45 t4.next = null;
23 }		46 }
		47 }

Fig. 1. The running example of this paper. The statements related to points-to analysis are highlighted.

from *list*1, *list*2 and *list*3 under the context 1, 2, 3 respectively and, assigning  $p_1$  to  $p_2$  is a one-to-one mapping, we conclude that  $(p_2)_y \mapsto o_x^{35}$ ,  $\forall x \in [1, 4)$ , y = x.

Second,  $gList = t1^8$  is also independently evaluated under three contexts. The difference is that gList is a global variable and it has only one version. Therefore, **all the versions of** t1 **are assigned to the single version of** gList, written as  $gList_1 = t1_x$ ,  $\forall x \in [1, 4)$ . This is a *many-to-one mapping* between t1 and gList, it can be interpreted as a horizontal segment  $y = 1, x \in [1, 4)$ .

The tricky case is the dereference assignment  $p_2.next = gList$ . Since we always refer to the same version of gList and it points to all three versions of  $o^{12}$ , evaluating  $p_2.next = gList$  would introduce nine new points-to relations between  $o^{35}.next$  and  $o^{12}$ . These relations are established in two steps. First, since gList would assign to all versions of  $p_2.next$  and  $(p_2)_y \mapsto o_x^{35}$ ,  $\forall x \in [1, 4), y = x$ , hereby we conclude gList assigns to all versions of  $o^{35}.next$ . After the replacement of  $p_2$  to  $o^{35}$ , we can reformulate the assignment as  $o^{35}.next_y = gList_1$ ,  $\forall y \in [1, 4)$ . This is an one-to-many mapping. Geometrically, we interpret the assignment as a vertical segment  $x = 1, y \in [1, 4), y \in [1, 4)$ . Algebraically, this is a many-to-many mapping and, geometrically, we can encode these relations as a rectangle where  $1 \le x \le 3$  and  $1 \le y \le 3$ .

By encoding the pointer assignments and points-to facts geometrically, we can use the corresponding algebraic operations to evaluate a large number of constraints under different contexts simultaneously. Let us demonstrate the idea by reiterating the evaluation with the encoded geometric shapes. Combining  $t1_y \mapsto o_x^{12}$   $(y = x, x \in [1, 4))$ and  $gList_1 = t1_x$   $(y = 1, x \in [1, 4))$ , we obtain  $gList_1 \mapsto o_x^{12}$   $(x \in [1, 4))$  because all ver-

<sup>&</sup>lt;sup>8</sup>The analysis is flow insensitive, hence the constraint evaluation order can be arbitrarily tuned.

ACM Transactions on Embedded Computing Systems, Vol. V, No. N, Article A, Publication date: January YYYY.



Fig. 2. The hierarchical call graph.

sions of  $o^{12}$  are passed to gList via all versions of the intermediate variable t1. Again, applying  $(p_2)_y \mapsto o_x^{35}$   $(y = x, x \in [1, 4))$  to  $p_2.next_y = gList_1$   $(y \in [1, 4))$ , we initialize the dereference assignment as  $o^{35}.next_y = gList_1$   $(y \in [1, 4))$ . Finally, we evaluate the initialized assignment and obtain  $o^{35}.next_y \mapsto o_x^{12}$   $(1 \le x \le 3, 1 \le y \le 3)$ . Computing these points-to relations geometrically only evaluates the corresponding constraints *once* independent of the number of contexts.

The discussion above illustrates the key concepts of the geometric encoding: Numbering the contexts, encoding the program statements, reasoning the encoded facts using geometric operations. Next, we describe these steps in detail and prove the soundness of our new points-to analysis.

## 3.2. Contexts Naming

Part 1 of Algorithm 1 explicitly constructs the cloned call graph. However, this step is only a conceptual processing step because the cloned call graph size is too huge. *Contexts naming* aims at labeling the call-strings with natural numbers and, thus, compressing the cloned call graph by encoding the contexts involved between the caller and the callee. For example, in Figure 3, the call edge prepare  $\rightarrow$  createList is cloned to three edges. If we only consider the context numbers for these call edges, we write them as  $1 \rightarrow 1$ ,  $2 \rightarrow 2$  and  $3 \rightarrow 3$ . Because the context numbers involved in the call edges are consecutive, we concisely write it as a one-to-one mapping between two intervals:  $[1,4) \rightarrow [1,4)$ , referred to as the *context mapping*.

In this section, our main task is to build the context mapping for every call edge. We pass the hierarchical call graph  $G_h$  as shown in Figure 2 to Algorithm 2 to locally number the contexts for each function X by the integers 1, 2, ..., N. N is called the

A:10



Fig. 3. The cloned call graph.

A	lgorit	hm	<b>2</b>	Num	ber t	he	static	cal	ling	pat	hs
---	--------	----	----------	-----	-------	----	--------	-----	------	-----	----

1: **procedure** NAMECONTEXTS(G<sub>h</sub>: Hierarchical Call Graph) for all function X do 2: 3: csize(X) = 0end for 4: csize(SuperMain) = 15: for all node X in  $G_c$  in topological order do 6: for all acyclic call edge  $X \to Y$  do 7:  $callmap[X \rightarrow Y] = csize(Y) + 1$ 8: 9: csize(Y) = csize(Y) + csize(X)end for 10: end for 11: for all none representative node X of SCC do 12:csize(X) = csize(rep(X))13:end for 14:for all cyclic call edge  $X \to Y$  do 15:  $callmap[X \to Y] = 1$ 16: end for 17: 18: end procedure

*context size*, denoted by csize(X), which is the number of acyclic call paths on  $G_h$  from **SuperMain** to X. The term *context bar*, symbolized as [1, csize(X)], describes all the contexts of function X in an interval form. The *context mapping* between two functions for a callsite  $X \to Y$  is captured by  $callmap[X \to Y] = offset$ , which maps a context of X to a context of Y by adding *offset* to that context of X. For example, we have  $callmap[X \to Y] = U$ , meaning that for each context  $c \in [1, csize(X)]$  of X, we map it to the context c' of Y, where c' = c + U.

Algorithm 2 visits all the functions in the topological order and obtains the first context of the context mapping for each call  $X \to Y$  by adding one to csize(Y) (Line 8). We then increase csize(Y) by csize(X), which determines the number of contexts mapped from X to Y (Line 9). In this way, the contexts induced by  $X \to Y$  are consecutively



Fig. 4. A pictorial explanation of building the context mappings for the call edges. The symbols  $X_1 - X_5$  and Y are functions, the arrows represent the context mappings.

located on the context bar of Y, which is crucial for us to use the simple regular geometric shapes for encoding the mapping relations. As an example shown in Figure 4, the contexts induced by the call  $X_3 \rightarrow Y$  are [15, 22), which is a consecutive interval. The cyclic call edges are still handled context insensitively (Lines 12–14), where the function rep(X) retrieves the representative of the SCC the function X belongs to. This simple strategy not only affects the SCC call edges, but also influences all of the calls to the functions in the SCC from outside, making a large part of the code context insensitive. We will generalize Algorithm 2 to handle the SCC more precisely in Section 3.5.

**Example**. The result of applying Algorithm 2 to our running example is given in Figure 1. The *callmap* content is drawn on the call graph edges, where  $[1, 2) \rightarrow [3, 4)$  of the third call edge (*a.k.a* e3) from main to prepare stands for *callmap*[e3] = 3.

## 3.3. Geometric Encoding System

Recall Algorithm 1, it builds the abstract domain  $\mathbb{AD}$  and constraints  $\mathbb{C}$  in Part 2 and Part 3. In this section, we show how to compress the abstract domain and generate the compressed form of constraints. After this step, all the statements that manipulate the pointers are encoded. Then, we introduce the inference rules that are used to interpret the semantics of the constraints in encoded form. This knowledge is prepared for the main points-to analysis algorithm given in Section 4.

3.3.1. Constructing the initial encoding. The initial encoding aims to build the encoded representation of the program facts. In Java, we only care about the pointer assignments (p = q), object allocation (p = new Object()) and the pointer dereferences (p.f = q and p = q.f). To ease our analysis, we assume the initial encoding is built on the static single assignment form <sup>9</sup> of the program.

**Pointer Assignments.** We first consider the assignment q = p, where both p and q are local to function X. The assignment has the semantics that p under the  $i^{th}$  context only assigns to q under the  $i^{th}$  context, written as  $q_i = p_i, \forall i \in [1, csize(X)]$ . Using the context numbers of p and q as the coordinates, all the points (i, i) on a plane essentially form a diagonal segment. To encode this fact, a 5-tuple representation (p, q, 1, 1, csize(X)) is provided to concisely name the assignments  $q_1 = p_1, q_2 = p_2, \ldots$ ,

 $<sup>^9 {\</sup>rm The}~\Phi$  function can be decomposed into several pointer assignments, hence we do not discuss it in this paper.

 $q_{csize(X)} = p_{csize(X)}$ . This expression faithfully captures the *one-to-one mapping* between p and q.

The function *callmap* computed by Algorithm 2 is leveraged if p and q, local to different functions, *e.g.*, X and Y, are involved in an inter-procedural assignment. In the case of parameter passing via the function call  $X \to Y$ , we let  $K = callmap[X \to Y]$ , the context sensitive form of the assignment q = p is expressed as (p, q, 1, K, csize(X)). Correspondingly, in the case of function return, the context sensitive version of q = p is encoded as (p, q, K, 1, csize(X)).

The assignments with globals are the major source of sophistication because globals are modeled context insensitively. For instance, the assignment g = p, where g is a global, means that all versions of p assign to the singleton version of g. This is a *many*-to-one mapping and its geometric interpretation is a horizontal segment. Conversely, the assignment p = g is a one-to-many mapping that represents a vertical segment. More complicated case occurred in the example g = p followed by q = g where q is another local. The relation between p and q is all versions of p assign to all versions of q, which forms a many-to-many mapping, figured as a rectangle.

We now define our symbols formally:

**DEFINITION 6.** The geometric encoding compresses the context sensitive points-to and assigns-to relations as a set of geometric descriptors  $(V_1, V_2, x_1, y_1, L_1, L_2)$   $(L_2$  is optional), abbreviated as  $E_{V_1/V_2}$ . The first two terms  $(V_1, V_2)$  is an **interpreter**, designating the variables involved in this encoded mapping relation. The following 4-tuple  $(x_1, y_1, L_1, L_2)$  or 3-tuple  $(x_1, y_1, L_1)$  is called **geometric extension**. It describes how the variable  $V_1$  is mapped to  $V_2$ , and we denote it  $[V_1, V_2]_{ext}$ .



Table I. Graphical explanation of the encoding tuples.

We treat the horizontal and the vertical segments as two special cases of rectangle. Hence, the geometric extension is either a diagonal segment or a rectangle. Formally, we define our encoding in the algebraic form as follows and show the geometric interpretation in Table I:

**DEFINITION** 7. The 3-tuple and 4-tuple geometric extension describes the following two geometric figures respectively:

**The diagonal segment** y = x + b. It indicates the one-to-one mapping which constantly offsets by b from X values to Y values in order. We encode this segment by a 5-tuple:  $(V_1, V_2, I_1, I_1 + b, L)$ , where  $[I_1, I_1 + L)$  is the range of X and  $[I_1 + b, I_1 + b + L)$  is the range of Y;

The rectangle bounded by four lines  $x = b_1$ ,  $x = b_2$ ,  $y = b_3$ ,  $y = b_4$  where  $b_1 < b_2$ and  $b_3 < b_4$ . It represents the many-to-many mapping, i.e., every value of X are mapped

to multiple values of Y. We encode the rectangle by a 6-tuple:  $(V_1, V_2, b_1, b_3, b_2-b_1, b_4-b_3)$ , where  $[b_1, b_2)$  and  $[b_3, b_4)$  are the ranges for the width and the height of the rectangle.

**Object Allocation**. Encoding the object allocation statements (p = new Object()) is very similar to encoding the pointer assignments. We first name the new expression (*e.g.* **new** *Object()*) by an unique name 0, then we treat 0 as a local if this new expression is written in a function, or a global if it appears in a static initializer for a class. Next, we encode p = 0 in the same way as an assignment statement (assume both p and 0 are local to function X), and obtain the outcome (p, o, 1, 1, csize(X)). Comparing to the encoded assignment, the only difference is that p is written as the first term. Moreover, all the generated points-to relations during the analysis are also represented in this form.



Fig. 5. The graphic form of the initial encoding.

**Pointer Dereference Assignments**. The pointer dereference statements include the *load* and the *store* statements that access the instance fields. For example, q.faccesses the f field of the object pointed to by q. The geometric extensions of the store statement q.f = p and the load statement q = p.f are exactly the geometric extension of its corresponding simple assignment q = p. This is because the encoded deference assignment is solely used to compute the geometric extension of the assignment to (from) instance field o.f = p (or q = o.f), by replacing q.f (or p.f) with  $q \mapsto o$  (or  $p \mapsto o$ ). We call this process *instantiation*.

**Example**. The initial encoding of our running example is included in Figure 5, where the geometric extensions are drawn on the arrows. Taking the assignment gList = t1 as an example, we encode it as (t1, gList, 1, 1, 3, 1), meaning all versions of t1 under the contexts [1, 1 + 3) are assigned to the single version [1, 1 + 1) of

gList. The points-to relation  $(t1, o^{12}, 1, 1, 3)$  is also represented in assignment form, for illustration purpose.

**Soundness**. The soundness of the initial encoding means all the possible assignments collected in Algorithm 1 are encoded. Of course, the subsequent reasoning will produce unsafe result if some facts are missed.

LEMMA 3.1. The initial encoding summarizes all the possible assignments under the full context sensitivity.

PROOF. We only show that the encoding of the simple pointer assignment statement p = q is sound, the other kind of statements can be proved using the same argument. First of all, the syntactically identical local variables under different contexts do not interfere each other. This is because a local variable is freshly allocated each time its enclosing function executed. Therefore, the  $i^{th}$  copy of q can only update the  $i^{th}$  copy of p in the assignment p = q, where p and q are two local variables inside the same function. Our *one-to-one* mapping representation faithfully encodes this fact, thus, it does not miss any assignments.

In the interprocedural assignment induced by the call  $X \to Y$ , if the  $i^{th}$  context of X can invoke the  $j^{th}$  context of Y, we should add an assignment  $p_j = q_i$ . Since the *callmap* function computed in Algorithm 2 captures all the context mappings induced by the function call, all the interprocedural assignments are encoded.

Since the global variable has only one version in our abstract domain (Section 2.1), in the assignment p = g (g = p) where g is a global variable, all versions of p read (write) the value from (to) the same version of g. Therefore, our encoded facts contain all the assignments involving the global variables.

From our analysis, we know the initial encoding is a sound summarization of all possible assignments under the full context sensitivity model.  $\Box$ 

3.3.2. Reasoning with the geometric encoding. The assignment constraint is reasoned under the fusion operator  $\circ$ : Given the geometric extensions of the relations p assigns to q and p points-to o, we compute the geometric extension of the relation q points-to o. For example, we have  $(t1, o^{17}, 1, 1, 3)$  and (t1, gList, 1, 1, 3, 1), the fusion result is  $(gList, o^{17}, 1, 1, 1, 3)$ . This is obtained in four steps. First, we extract the context ranges of pointer  $t1^{-10}$  in the points-to and assigns-to figures, which are [1, 4) and [1, 4). Second, we intersect these two ranges and obtain the common section [1, 4). We call this step clipping. Third, we compute the intervals of gList and o respecting to the interval [1, 4) of p. In our case, they are [1, 2) and [1, 4). Finally, we compute the mapping relation between gList and o, and we call this step expanding. Since all  $p_i$  ( $i \in [1, 4)$ ) assign to the single copy of gList, it is a many-to-many mapping and encoded by a rectangle:  $(gList, o^{17}, 1, 1, 1, 3)$ .

Instantiating the complex constraint can be performed in the same way as evaluating the pointer assignment, represented by the operator  $\bullet$ . For example, we instantiate the store constraint (nxt, t3.next, 1, 1, 3) against the points-to fact  $(t3, o^{35}, 1, 1, 3)$ . We also need four steps of calculation discussed above, but this time the agent pointer is t3 and its intersected interval is [1, 4) in the clipping step. After mapping the intersection interval to the variables nxt and  $o^{35}$ , we obtain  $(nxt, o^{35}.next, 1, 1, 3)$ .

We depict the inference rules for the pointer assignments in Table II <sup>11</sup>. Each inference rule contains three mutually perpendicular planes, and the input shapes are given in the planes < p, o > and < p, q >. We deduce the shape on the plane < q, o > by

<sup>&</sup>lt;sup>10</sup>Since t1 appears in both tuples, it is called the *agent pointer*.

<sup>&</sup>lt;sup>11</sup>Because the complex constraints instantiation are exactly the same, we thereby elude the details.

ACM Transactions on Embedded Computing Systems, Vol. V, No. N, Article A, Publication date: January YYYY.

Table II. Assignment Inference Rules. For each picture, the figures on the planes < p, o > and < p, q > are given as input, describing the p points-to o relation and the p assigns-to q relation. The generation steps of the figure on the plane < q, o > are implicitly stated by the dashed lines, which stand for the clipping and expanding operations.



the clipping and expanding steps explained before, which are rendered by the dashed lines. The application of these rules to our discussed instances in the previous two paragraphs are given in Table III. And the points-to solution of repeatedly applying these rules are given in Figure 6. From Table II, we conclude that  $\langle S, \circ \rangle$  and  $\langle S', \bullet \rangle$ are two magma algebraic structures because the computation is closed under the operators  $\circ$  and  $\bullet$ , where S and S' are sets of diagonal segments and rectangles. This result indicates the inherit membership complexity of geometric encoding is quite low, because only two simple geometric figures are involved. Moreover, the soundness of the inference rules can be verified easily. We prove it in Lemma 3.2.

## LEMMA 3.2. The inference rules for $\circ$ and $\bullet$ are sound.

PROOF. We only prove the soundness for the operator  $\circ$  and the argument is applicable to  $\bullet$  as well. The correctness of the clipping step is intuitive, of course we only need to consider the common contexts of the agent pointer. The expanding step infers the shape of the result figure, of which the correctness is discussed as follows:

1. The input  $[p, o]_{ext}$  and  $[p, q]_{ext}$  are both diagonal segments. Because, for any  $p_i$ , we know there is only one  $o_j$  satisfying  $p_i \mapsto o_j$  and only one  $q_k$  satisfying  $p_i \rightsquigarrow q_k$ , we





Fig. 6. The graphic form of the points-to solution. The result of  $p_2$  is the union of the  $p_1$  and  $p_3$ .

conclude that  $o_j$  is only pointed-to by  $q_k$  in the inference result. Therefore,  $[q, o]_{ext}$  is a diagonal line;

2. The input  $[p, o]_{ext}$  is a diagonal segment and  $[p, q]_{ext}$  is a rectangle. For any  $o_j$ , it is pointed-to by only one  $p_i$ . However,  $p_i$  is assigned to all versions of q. We know that all versions of q point to  $o_j$ . Thus,  $[q, o]_{ext}$  is a rectangle;

3. The input  $[p, o]_{ext}$  is a rectangle and  $[p, q]_{ext}$  is a diagonal segment. In this situation,  $p_i$  points to all versions of o, and  $p_i$  is exclusively assigned to  $q_k$ , hence,  $q_k$  points to all versions of o. Therefore,  $[q, o]_{ext}$  is a rectangle;

4. The input  $[p, o]_{ext}$  and  $[p, q]_{ext}$  are both rectangles. This inference rule can be seen as a special case of inference rule 2 or 3. Thus,  $[q, o]_{ext}$  is a rectangle.

In sum, the inference rules for  $\circ$  and  $\bullet$  are sound.  $\Box$ 

## 3.4. Characteristics of Geometric Encoding

The geometric encoding overall offers the higher compression capability and precision fidelity than the state-of-the-art non-BDD based points-to analysis EPA [Xu and Roun-

tev 2008]. Without the globals, our encoding is as compact and precise as EPA. This property can be demonstrated through an example of two functions, X and Y, that share a lowest common ancestor function Z on a SCC-condensed graph. If a pointer p in X points to an object o created in Y, EPA represents the fact by a 4-tuple  $(p, \xi_1, o, \xi_2)$ , where the symbols  $\xi_1$  and  $\xi_2$  are the call paths to X and Y descending from Z. In our representation,  $(p, \xi_1, o, \xi_2)$  is encoded as  $(p, o, i_p, i_o, csize(Z))$ , denoting that the csize(Z)number of p from the context  $i_p$  points to the csize(Z) number of o from the context  $i_o$ . These two encodings have no difference except the representation of the contexts. Therefore, both encodings have the same compression capability and precision fidelity.

However, since the EPA algorithm does not clone the objects o pointed to by the globals, it causes two problems. First, it cannot compress those points-to facts related to o, because o is treated context insensitively and no common call string prefixes can be exploited. Second, as the consequence of the context insensitivity, the precision is degraded. Intuitively, consider our sample code in Figure 1. Since *gList* points to some versions of  $o^{12}$ , the EPA algorithm directly makes  $o^{12}$  insensitive, making q2 an alias to both *list1* and *list3*. This is because the single version of  $o^{12}$ .next points to all versions of  $o^{35}$  under the EPA approach, for which our encoding correctly concludes that the different versions of  $o^{12}$ .next point to different versions of  $o^{35}$ . Therefore, in the presence of the globals, our encoding is more compact and precise than EPA.

Another noteworthy characteristic of our geometric encoding is that the evaluations of all kinds of constraints are always O(1). While BDD can also perform a group of assignments and instantiations in one operation, the complexity varies from O(1) to  $O(U^2)$  with no guarantee, where U is the maximum context size of all the functions. The is because the complexity of the relprod operation for computing the *relational* product of two BDDs is  $O(n_1n_2)$ , where  $n_1$  and  $n_2$  are the number of nodes of the input BDDs given to relprod, ranging from O(1) to O(U) ( $O(U) = O(2^{\log U})$ ).

### 3.5. Recursive Calls Revisited

Due to the unbounded number of contexts incurred by the SCCs, the full context sensitive analysis usually, including our algorithm presented so far, apply a 0-CFA model by contracting each SCC to a single node and computing the context insensitive results inside of the SCCs. This treatment causes to precision degradation because the points-to facts for the variables inside of these SCCs become imprecise and, the call edges between the non-SCC and SCC functions further exacerbate the degradation of the analysis quality by propagating the imprecise results to the whole program. This is problematic because Java programs tend to have large SCCs due to the imprecise call graph resolution.

Algorithm 2 produces a 0-CFA model for SCCs because, for any recursive call  $X \to Y$ , we restrict csize(X) = csize(Y) and  $callmap[X \to Y] = 1$ . With this treatment, for any other call edge  $callmap[X' \to Y] = s$ , every context  $i \in [s, s+csize(X'))$  of Y is also used as the image of the  $i^{th}$  context in the context mapping induced by  $X \to Y$ . Therefore, the return value of  $Y_i$  will be passed to both X and X',  $\forall i \in [s, s+csize(X'))$ .

The core idea of our remedy, the *blocking scheme*, is allowing the SCC members to have different number of contexts. Consider the partial call graph in Figure 7(a), we obtain its *1*-CFA model in Figure 7(b) as follows.

1. We still run Algorithm 2 first and suppose the functions A, B, E and k others in the SCC all have m contexts, the function D outside the SCC has n contexts (n < m).

2. For each function in the SCC, we re-calculate its context size. Taking function C as an example, it has k+3 incoming calls. Hence, we reassign  $csize(C) = (k+2) \times m+n$  and divide its context bar into k+3 blocks, where the first k+2 blocks have m contexts and the last one has n contexts.



a. Partial call graph involving SCC

b. 1-CFA abstraction for the SCC

Fig. 7. The *1*-CFA modeling of an SCC.  $A, \ldots, E, f_1 \ldots f_k$  are functions, the segments under which denote the context bars, and the lines denote the context mappings.

3. We map the call  $A \to C$  to the first block,  $B \to C$  to the second block, and so on. This step is finished by changing  $callmap[A \to C] = 1$ ,  $callmap[B \to C] = m + 1$ , ...,  $callmap[D \to C] = (k+2) \times m + 1$ .

After processing all the SCC functions in the way above, we build a new context function *bcf*. We say *bcf* treats the SCC functions in the 1-CFA manner because the input points-to information passed to C from different callsites (A, B, D, *etc.*) are processed without interference, and, if the points-to information is passed to E, it is merged again. Nevertheless, as confirmed by the experiments, our improved model for SCCs still gains non-negligible precision over the 0-CFA model.

We should also adapt the way of building initial encoding to the new context function *bcf*. As an example, we try to build the encoding for the parameter passing, p = q, incurred by the call  $C \to E$ . With *bcf*, we have csize(C) > csize(E), hence we cannot simply generate the constraint  $(q, p, 1, callmap[C \to E], csize(C))$ . Instead, we map all the blocks of C one-by-one on to E. This time, we build k + 3 mappings from C to E for the k + 3 blocks of C, hence we generate k + 3 constraints  $(q, p, 1, callmap[C \to E], m)$ ,  $(q, p, \mathbf{m}, callmap[C \to E], m), \ldots, (q, p, (\mathbf{k+2}) \times \mathbf{m}, callmap[C \to E], n)$ .

Let us apply our blocking scheme to the call graph in Figure 2 and illustrate the blocked mapping structure in Figure 8. In our example, the createList function is extended to two blocks, occupying the contexts [1, 4) and [4, 7) and used by prepare and setNext, respectively. The createList also calls the initialize function, of which the contexts are induced by the contexts [1, 4) of createList. For this callsite, we map both the context blocks [1, 4) and [4, 7) of createList to [1, 4) of initialize.

Finally, we prove the context function *bcf* is sound in Lemma 3.3. Notice that, this lemma is a constructive proof, it can also be used for translating any call-string to our integer representation of contexts.

LEMMA 3.3. The context function  $bcf : \mathbb{R} \to \mathbb{S}$  is sound.



Fig. 8. Blocking scheme illustration for our sample code. The dotted lines represent the context mappings introduced by the blocking scheme.

PROOF. We first prove that any runtime call-string  $e_1e_2\cdots e_K$  leading to the function U is mapped to a unique context of U. Our proof uses an auxiliary function  $\phi_X(e)$  for all the call edge  $e: Z \to X$ .  $\phi_X(e)$  is built as follows.

1. *X* is not a member of any SCC. In this case, we do not apply the blocking scheme to *X*, thus,  $\phi_X(e) = callmap[e]$ .

2. *X* is a member of any SCC. To compute  $\phi_X(e)$ , we modify the second step of our blocking scheme. Similar to csize(X), we maintain an array bsize(X) and initialize it to one. Then, we visit the incoming call edges of *X* one by one. When processing the edge  $e: Z \to X$ , we assign  $\phi_X(e) = bsize(X)$  and update bsize(X) = bsize(X) + csize(Z).

Now we translate the call-string  $e_1e_2 \cdots e_K$  starting at SuperMain to our integer context *i*. By our definition of SuparMain,  $e_1$  cannot be a recursive call, thus we directly assign  $i = callmap[e_1]$ . For the rest of the call edges,  $e_t : X \to Y$ ,  $1 < t \leq K$ , the context is transferred as:  $i = i - \phi(e_{t-1}) + callmap(e_t)$ . In this formula, the first term  $i - \phi(e_{t-1})$  computes the offset of the context *i* to the first context of its enclosing block. This result is required because, according to the blocking scheme, all the blocks of X are individually mapped onto Y. Thus, we should know the offset of *i* within its enclosing block. Next, we use the *callmap* function to help locate the first context *i* of U that the input call-string is mapped onto.

Combining the fact that the located context *i* is unique and every function instance can be mapped to a runtime call-string, we prove that every element in  $\mathbb{R}$  is mapped to an integer context in  $\mathbb{S}$ .  $\Box$ 

## 4. THE POINTS-TO ALGORITHM

The skeleton of our points-to algorithm is given in Algorithm 3, which only replaces the inference rules of the Anderson's analysis with our new rules given in Section 3.3.2. In summary, the anatomy of our points-to algorithm is as follows.

Lines 2–5: These lines are the preparation steps. The function constraintsDistillation eliminates the useless constraints that do not affect the pointers in the user's code, and the function offlineMerging identify the pointers that may have the same context sensitive points-to result. These two analyses are given in Section 4.1. The third function, buildInitialEncoding, creates the initial encoding for the program facts. The last initialization work is putting the pointers that are the receivers p at the allocation sites,  $p = \mathbf{new} \ Object()$ , into the worklist.

Algorithm 3 The Points-to Analysis Main Algorithm

```
1: procedure POINTSTOANALYSIS
        CONSTRAINTSDISTILLATION()
2:
       OFFLINEMERGING()
3:
       BUILDINITIALENCODING()
4:
       Worklist \leftarrow pointers have unprocessed points-to tuples
5:
6:
7:
        while Worklist \neq \emptyset do
           pick a pointer p from Worklist
8:
           GEOMETRICMERGING(p)
                                                       ▷ Must perform at the start of the loop
9:
10:
           for all newly added points-to relation E_{p/o} do
               for all store constraint E_{p/q.f} do

E_{p/o.f} = \text{INSTANTIATE}(E_{p/o}, E_{p/q.f})

if ADDEDGE(E_{p/o.f}) then

put o.f into Worklist
11:
12:
13:
14:
                   end if
15:
               end for
16:
               for all load constraint E_{q.f/p} do
17:
18:
                    E_{o.f/p} = \text{INSTANTIATE}(E_{p/o}, E_{q.f/p})
                   if ADDEDGE(E_{o.f/p}) then
19:
                       put p into Worklist
20:
                   end if
21:
               end for
22:
23:
            end for
24:
           for all pointer assignment E_{p/q} do
25:
               for all points-to relation \vec{E}_{p/o} do
26:
                   PROPAGATE(E_{p/q}, E_{p/o})
27:
                   if ADDPOINTS TO(E_{q/o}) then
28:
                       put q into Worklist
29:
                    end if
30:
31:
               end for
32:
            end for
        end while
33.
34: end procedure
```

Lines 9: Before processing a pointer p, we try to merge its points-to tuples (e.g. (p, o, 1, 2, 3)) and the assigns-to tuples (e.g. (p, q, 5, 6, 7)) for saving memory and speeding up the fixed-point convergence. This is achieved by finding the bounded rectangle of these small geometric figures (Figure 9). This operation is called geometric merging and the details are given in Section 4.1.

Lines 10–23: We instantiate the complex constraints taking p as the agent pointer (Section 3.3.2). The function addEdge called at line 13 adds the instantiated constraint to the PAG, if there does not exist a bigger geometric figure that completely covers the geometric figure of this new edge.

Lines 25–32: This piece of code uses the assignment inference rules (Table II) to propagate the points-to facts. addPointsTo adds a new points-to relation to the points-to solution and puts the receiver pointer q into the worklist, *if there does not exist a bigger geometric figure that covers the geometric figure of the new points-to relation*.

Our implementation of Algorithm 3 also employs the common acceleration techniques for Anderson's analysis such as the difference propagation and the prioritized

worklist [Pearce et al. 2003]. More details about the implementation can be found in Section 5.

#### 4.1. Optimizations

**Constraints Distillation**. The prevalent use of libraries in Java program inhibits the scaling of the points-to analysis. However, most of the time, only the precise points-to information for the user's code is necessary for the subsequent use. This observation is similar to Rountev's [Rountev and Ryder 2001]: A large part of the library code actually does not affect the points-to information of the pointers in user's code. It is also similar to the demand driven spirit that not all code is needed for computing the points-to information of a pointer [Sridharan and Bodík 2006]. Therefore, we distill the constraints before the points-to analysis in order to reduce the computation effort without precision penalty.

Our approach can be more precise than that of Rountev's [Rountev and Ryder 2001] because we have the whole program prior to the analysis. We first identify the pointers of which its points-to information is essential. The idea is, to obtain the points-to information of q, we only need the points-to information of p or o.f if they are assigned to q. This process is similar to searching the demanded constraints in Heintz *et.al*'s algorithm [Heintze and Tardieu 2001]. Specifically, we find out all the essential pointers by propagating marks on **the final assignment graph produced by the Anderson's analysis**, starting by marking the pointers appeared in the user's classes. Next, we distill the irrelevant constraints: q = p is irrelevant iff q is inessential. In this way, many pointers cannot obtain the points-to information after performing our algorithm. To compensate these pointers, we directly inject the points-to information computed by Anderson's analysis into their points-to results.

With the constraints distillation technique, our points-to algorithm can be configured as a demand driven analysis. The user can first specify the pointers that need more precise points-to information, then perform the distillation algorithm to obtain only the minimum set of constraints for evaluation. The points-to algorithm can also be easily incrementalized based on the distillation technique. As an example, we compute pts(p) and pts(q) in two queries. In the first query of computing pts(p), we need the points-to information of x, y, z. In the second query of computing pts(q), we need pts(x) and pts(u). Since pts(x) was fully computed, we can soundly reuse it in the second query.

**Offline Equivalent Pointers Merging**. Equivalent pointers are those that have the same points-to information at the end of the points-to analysis. Due to the temporary variables introduced by the intermediate representation and the flow insensitivity, many pointers have the same points-to results. Therefore, merging those equivalent pointers before the points-to analysis can help terminate the algorithm earlier.

In the area of context insensitive analysis, Rountev *et.al* first propose an algorithm to detect the equivalent pointers by mark propagation on the pointer assignment graph before the constraints evaluation [Rountev and Chandra 2000]. Their mark propagation method is then slightly improved by Hardekopf *et.al* with global value numbering [Hardekopf and Lin 2007b]. However, since the number of pointers is very large under the full context sensitivity model, using the geometric encoding based version of these algorithms is still expensive in time and memory.

Therefore, we employ a simple intraprocedural detection algorithm. Our algorithm merges a local pointer q with another local pointer p, if they both local to the same function and q = p is the only assignment that assigns to q. Under this condition, we have  $p_i \mapsto o_j \Leftrightarrow q_i \mapsto o_j$ ,  $\forall i \in [1, N]$ , where N is the context size of the enclosing



Fig. 9. Geometric Merging Illustration. Picture (a) shows the input figures, and picture (b) outlines the corresponding bounding rectangles.

function of p and q. This is because the segment (p, q, 1, 1, N) is the identity element for any magma structure  $\langle S, \circ \rangle$ . Therefore, our equivalent pointer merging technique is precision preserving.

**Geometric Merging**. Reasoning with the geometric encoding may produce quite a lot small geometric figures, due to the call edges between the functions with small context sizes. In our running example, the function prepare is called three times in main, and the return values are assigned to *list1*, *list2* and *list3*, then assigned to  $p_1$ , and finally to  $p_2$ . Along this way, we make three points-to tuples for  $p_2$ , representing the points-to information  $(p_2)_1 \mapsto o_1^{32}$ ,  $(p_2)_2 \mapsto o_2^{32}$  and  $(p_2)_3 \mapsto o_3^{32}$  (Figure 6). The problem is, when we evaluate  $p_3 = p_2.next$ , we will create three instantiated assignments. This, in turn, reduces the compression efficiency and increases the analysis time.

To obtain good performance, we merge the small geometric figures into a larger one. The merged geometric figure is the bounding rectangle of all the shapes described by the extensions before merging, as shown in Figure 9. The merging plays as a widening operator [Cousot and Cousot 1977] to speedup convergence. Since the merged figure covers all the figures before merging, it is a sound approximation.

Merging, of course, sacrifices the precision. To select the pointer to merge, we set two *fractional parameters*, referred to as  $\delta_1$  and  $\delta_2$ , to limit the number of geometric extensions every interpreter tuple  $(V_1, V_2)$  owns <sup>12</sup>. In Algorithm 3, **immediately after fetching a pointer**, p, **from the worklist**, we check the number of geometric extensions that every points-to tuple (p, o) and flow edge (p, q) own. We merge all the extensions into a single rectangle if (p, o) (or (p, q)) has more than  $\delta_1$  (or  $\delta_2$ ) geometric extensions, and at least one of which is newly owned by (p, o) (or (p, q)), where the term *newly* indicates that geometric extension is not propagated, in the vocabulary of difference propagation [Pearce et al. 2003].

#### 4.2. Soundness

THEOREM 4.1. Algorithm 3 is a sound points-to analysis.

PROOF. The soundness of the points-to Algorithm 3 has two parts: it always terminates and it always produces safe solution. The termination is because the context for every pointer and object is finite, thus, the number of possible geometric extensions are finite. In the case of repeated geometric extensions are generated, calling addEdge

 $<sup>^{12}</sup>$ If we have an encoded points-to fact (p, o, 1, 1, 2), we say (p, o) owns the extension (1, 1, 2).

ACM Transactions on Embedded Computing Systems, Vol. V, No. N, Article A, Publication date: January YYYY.

(Line 13, Line 19) and addPointsTo (Line 28) will return false. Therefore, the worklist will finally be empty and the loop will exit.

Without the geometric merging, the fixed point solution is safe because according to Lemma 3.1, Lemma 3.2 and Lemma 3.3, Algorithm 3 can be seen as an encoded version of Algorithm 1. However, because we strictly let the geometric merging perform at the start of the while loop, the fixed point solution is still safe. This is because the fixed solution is a stationary state that for every constraint in the assignment graph, if we have  $q \supseteq p$ , then  $\forall i, j, \exists k, p_i \mapsto o_j \Rightarrow q_k \mapsto o_j$ . If,  $p_i \mapsto o_j$  is generated by the geometric merging and the merging happened before the evaluation of the constraints in which p is on the right hand side, we know that  $p_i \mapsto o_j$  must be propagated at Line 27. Thus, the constraints are always interpreted according to their semantics, *i.e.*, the fixed solution computed by Algorithm 3 is safe.  $\Box$ 

# 5. IMPLEMENTATION

Unlike the Whaley-Zhu's algorithm [Whaley and Lam 2004; Zhu and Calman 2004] and the Xu's algorithm [Xu and Rountev 2008], the geometric encoding based pointsto analysis allows almost all the mature implementation and optimization techniques designed for Anderson's analysis directly reusable in our algorithm. The constraints distillation and offline equivalent pointer merging techniques, presented in Section 4.1, are two good examples. In this section, we first dive into the coding level to study the data structures for the geometric descriptors and the priority worklist. These two components are central to the implementation of the difference propagation and topology based propagation [Pearce et al. 2003], both are proven to heavily influence the performance of Anderson's style algorithm. We also discuss the problem of dealing with the contexts beyond  $2^{63}$  – because we use the java *long* type to manipulate the contexts – and the issue of cleaning the spurious points-to information for the special *this* pointer.

However, the technique of online cycle detection, proven very efficient for C programs [Fähndrich et al. 1998], is omitted in our implementation. This is because Java is a type safe language, the benefit of using the type filter in the points-to analysis can be more significant than the online cycle detection [Lhoták 2002; Sridharan and Fink 2009]. Meanwhile, the type filter is not fully compatible with cycle detection and it is not entirely clear how to keep the benefit of the type filter while incorporating the cycle elimination. Therefore, it is not cost benefit to implement the online cycle detection in our points-to engine.

#### 5.1. Managing Geometric Descriptors

We have four tasks for the manager of the geometric descriptors. First, in the evaluation of reasoning rules (Section 3.3.2), it can efficiently report all the existing figures that have non-empty intersection with the input figure on the agent pointer. Second, given a new geometric figure, it can quickly confirm whether it is covered by the combination of existing figures or not. Third, if the new figure is not totally covered, we should conversely find out all the existing figures that are covered by the new figure. Fourth, it is able to seamlessly combine the existing figures to form a bigger one if possible. For example, in the solution of example (Figure 6), the points-to tuples for  $p_1$ can be combined into  $(p_1, o^{32}, 1, 1, 3)$  without precision loss.

Building a data structure that can efficiently solve our problem is hard, especially we additionally require the data structure to be memory friendly. The data structure, such as R-tree, is only efficient for the situation with massive data and more tree modifying (insert and delete) operations than tree searching operations. It is not the case for our problem setting because the number of geometric figures owned by each pointer rarely exceeds one hundred, and most of them only have less than ten figures,

A:24

according to our observation. The tree modifying operations are frequently performed, because we must perform at least one insert or delete operation each time the addEdge or the addPointsTo function is called. The other choices, such as the 2D interval tree, is also not memory friendly in our case. Because, the size of the tree is proportional to the maximum context size among all the functions.

Therefore, after our extensive experiments, we finally choose the linked list to store the segments and rectangles. In this way, the first and the third tasks above are only a linear scan through the linked list. The requirement for the second task is also simplified that, for a queried figure, we only determine if there exists figure that can fully cover the queried figure. The fourth job can be partially handled by the technique geometric merging (Section 4.1). And we do not perform additional actions because it already works well. To support the differential propagation, we place all the newly inserted figures at the head of the lists and label them by *new*. Therefore, the function propagate (Line 28 in Algorithm 3) only visits the figure labeled *new*.

### 5.2. Priority Worklist

As pointed out by the previous work [Pearce et al. 2003], the performance of Anderson's analysis is significantly influenced by the worklist selection strategy (Line 8 of Algorithm 3). Of course, the best strategy is that we always select the first node topologically <sup>13</sup> in the worklist and maintain the topological order while the assignment graph changes. However, this is too expensive to implement in practice. We develop a hybrid approach inspired by the HCD technique proposed by Hardekopf et.al [Hardekopf and Lin 2007a]. Precisely, we first build an offline assignment graph using the constraints extracted from the code. For each p = q, we add an edge  $q \rightarrow p$ . For each q = p.f, we add an edge  $p.f \rightarrow q$ . For each p.f = q, we add an edge  $q \rightarrow p.f$ . Note that, the symbol p.f is a wildcard for all the instance fields  $o_1.f, o_2.f, \ldots$ , where  $o_1$ ,  $o_2, \ldots$  are the unknown objects that are pointed to by p. Next, we topologically sort the assignment graph by first collapsing the cycles, then, every node in the graph obtains a topological value. Later during the points-to analysis, if we find that p points to o, we immediately let the topological value of o.f equal to that of p.f. However, since we have cycles in the graph, many pointers may gain the same topological value, which is insufficient for sorting the pointers in the worklist. To complement the deficiency, we apply the LRU (Least Recently Used) strategy [Pearce et al. 2003]. In short, we maintain a global time stamp. Each time a pointer is pushed into the worklist, we increase the time stamp by one and assign it to that pointer. At last, the worklist strategy is that the pointer owning the least topological value and the least time stamp is the next one to be selected. Thus, it guarantees the total ordering of the pointers.

# 5.3. Contexts Beyond $2^{63}$

We use Java primitive type *long* to define the *csize* function and implement the geometric descriptors. However, *long* only supports up to  $2^{63} - 1$  (*a.k.a* **MaxLong**) contexts, which is insufficient for the large programs. In case of csize[Y] + csize[X] exceeds **MaxLong** (Line 9 of Algorithm 2), we mark the edge  $X \to Y$  as *exploded call edge* and randomly assign a value to  $callmap[X \to Y]$  between 0 and MAX\_LONG – csize[X], in order to minimize the interferences among different exploded call edges to Y.

We simplify the blocking method as described in Section 3.5 to handle the exploded call edges encountered in the blocking scheme. First, we only remap the recursive calls to the new positions, keep the map-to positions of the non-recursive calls remain the same. Second, since only the recursive calls are remapped, the context size of a newly created block of function X in an SCC must be the value csize[X] before applying the

<sup>&</sup>lt;sup>13</sup>The topological order is defined on the pointer assignment graph.

ACM Transactions on Embedded Computing Systems, Vol. V, No. N, Article A, Publication date: January YYYY.

blocking scheme. Therefore, in our implementation, we will construct  $N_{in} + 1$  blocks for the function X if no call edges are exploded and, all the none recursive calls are mapped to the first block, where  $N_{in}$  is the number of incoming recursive calls. Otherwise, if  $X \to Y$  is an exploded edge, we do not create a new block for it. Instead, we randomly reuse one of the existing blocks (at least the first block must be there).

## 5.4. Points-to filter for this pointer

Many Java methods retrieve the contents of an object through *this* pointer. Thus, better precision for *this* pointer is absolutely desirable. The *this* pointer obtains its points-to information from the base pointers at the callsites which invoke its enclosing function. For example, we have a callsite **p.foo(...)**, it induces an assignment *this*<sub>foo</sub> = p. However, due to the polymorphism, we may statically create many call edges to the different implementation of the function **foo**. Suppose we have  $pts(p) = o_1, o_2, o_3$ , where the objects  $o_1$ ,  $o_2$ , and  $o_3$  are in the types A, B, and C respectively. The type A is a super class of B and B is a super class of C, and, all three types provide their own implementation of **foo**. In this situation, the type filter cannot prevent from passing the objects  $o_2$  and  $o_3$  to the *this* pointer for the **foo** function implemented in type A. In order to eliminate such spurious assignment, when a *this* pointer received an object o, we check if we can resolve to the enclosing function of this *this* pointer through the type of o. If we cannot, we remove o from the points-to set of *this*.

#### 6. EVALUATION

The goal of the experiment is to examine the performance and precision characteristics of our algorithm. We implement our geometric encoding (Geom) points-to algorithm in the Soot framework <sup>14</sup>. We choose the 1-object-sensitive algorithm [Milanova et al. 2005] implemented in Paddle [Lhoták 2006] as the representative of the existing context sensitive analysis, because Lhoták *et.al* [Lhoták and Hendren 2008], beside us, also consider it providing the best tradeoff between precision and analysis efficiency. We use the worklist based implementation of the 1-object-sensitive analysis and refer to it 1-obj-W. We do not use the BDD version because it is too slow and always crashes due to unknown reason, which is also observed by Xu *et.al* [Xu and Rountev 2008] and in our previous experiment [Xiao and Zhang 2011]. We compare the performance of Geom and 1-obj-W by measuring their time and memory usage. And we assess their precision by the average points-to set size metric and three clients: virtual call resolution, static casts analysis, and alias analysis, all of which are widely used in the points-to analysis literature.

Our reference implementation has been accepted by the Soot maintenance team, and it was available since version 2.5. Any interested readers can download the software <sup>15</sup> and use it in your own work.

#### 6.1. Experimental Setting

**Environment.** We use the nightly built Soot as the analysis engine and employ Sun JDK 1.4 to 1.6 as the program analysis base library. To the best of our knowledge, this is the first time using JDK 1.5 or above versions in the context sensitive points-to analysis. Soot itself is powered by JRokit 28.1 running in the *server* mode and parametrized for minimum JVM latency, *i.e.*, the minimum/maximum heap size are both set to 30GB in order to reduce the JVM memory management time (memory fetching and garbage collection). All the evaluation data are collected on a 64-bit

ACM Transactions on Embedded Computing Systems, Vol. V, No. N, Article A, Publication date: January YYYY.

A:26

<sup>&</sup>lt;sup>14</sup>http://www.sable.mcgill.ca/soot/

 $<sup>^{15}</sup>$ A patch has been submitted to fix some known bugs in version 2.5. Please download the newest version or use the nightly built version whenever available.

	Program	LOC	Constraints	Max Contexts	Methods	Max SCC	JDK	Source
	jflex	141.3K	63.2%	$3.9 \times 10^9$	10533	3814	1.4	Ver 1.3.5
	soot	103.3K	74.2%	$9.3  imes 10^{11}$	7254	1718	1.4	Ashes
e	sableCC	152.1K	68.2%	$4.7 \times 10^{10}$	12179	3710	1.4	Ashes
Suit	ps	103.1K	70.1%	$5.2 \times 10^{10}$	8039	2106	1.4	beta050224
	antlr05	85.6K	67.6%	$1.0 \times 10^{12}$	5708	1675	1.4	beta050224
3ms	bloat05	129.0K	75.2%	$1.0 \times 10^{11}$	8187	2575	1.4	beta050224
01	pmd05	90.2K	66.3%	$\geq 2^{63}$	6094	1610	1.4	beta050224
	jedit1	270.8K	70.5%	$7.6 \times 10^8$	17548	7993	1.4	Ver 4.0
	megmek	303.3K	71.0%	$1.0  imes 10^{11}$	17379	7552	1.4	Ver 0.30.0
	antlr06	153.1K	68.2%	$1.3 \times 10^{14}$	9831	2568	1.6	2006MR2
	bloat06	190.0K	73.2%	$7.3 \times 10^{12}$	11745	3414	1.6	2006MR2
te	jython	188.0K	76.0%	$5.3 \times 10^{9}$	12729	4259	1.6	2006MR2
Sui	luindex	145.8K	67.3%	$4.2 \times 10^{9}$	9593	2550	1.6	2006MR2
lle	pmd06	173.4K	69.2%	$\geq 2^{63}$	11103	2612	1.6	2006MR2
lide	xalan	138.2K	53.6%	$1.8 \times 10^9$	9106	2548	1.6	2006MR2
2	chart	384.4K	66.6%	$3.6 \times 10^{13}$	25083	7381	1.6	2006MR2
	eclipse	163.0K	69.4%	$7.5 \times 10^9$	10372	2913	1.6	2006MR2
	hsqldb	139.1K	66.8%	$2.2 \times 10^9$	9104	2565	1.6	2006MR2
e	batik	411.9K	66.8%	$\geq 2^{63}$	26193	8398	1.6	9.12bach
Suit	sunflow	334.0K	63.3%	$\geq 2^{63}$	22520	7051	1.6	9.12bach
ge S	tomcat	363.1K	65.9%	$\geq 2^{63}$	25232	8027	1.6	9.12bach
arg	h2	278.5K	76.9%	$\geq 2^{63}$	16834	6301	1.6	9.12bach
	jedit2	414.2K	70.1%	$4.6  imes 10^{10}$	26470	9985	1.5	Ver 4.3.2

Table IV. Summary of the benchmarks. The LOC column counts the lines of code for the Jimple IR. The Constraints column shows the percentage of constraints that is really feed to the geometric points-to analysis.

machine with an Intel Xeon 3.0G processor running Linux 2.6.32.

**Benchmarks.** Our benchmarking programs are divided into three suites. The first two suite contain a set of small and middle sized programs, analyzed with JDK 1.4 and 1.6. These two suites are mainly used for the comparison to 1-obj-W, because 1-obj-W is running out of memory on our largest suite. The last suite is majorly used for the self-assessment of scalability. All the basic information of these three suites can be found in Table IV. The labels Ashes, beta050224, 2006MR2 and 9.12bach in the **Source** column indicate these programs are selected from the Ashes <sup>16</sup>, Dacapobeta050224, Dacapo-2006-10-MR2 and Dacapo-9.12bach benchmark collections. We select all the programs from the Dacapo-2006-10-MR2 suite except the cases *lusearch* and *fop*, because SPARK throws an exception for unknown reasons. The programs selected from Dacapo-9.12bach are all processed by feeding the default input to Tamiflex for the reflection resolution [Bodden et al. 2011].

**Parameters.** The initial call graph used to bootstrap our context sensitive analysis is computed by SPARK [Lhoták and Laurie 2003]. SPARK is running without offline simplification (*simplify-offline* and *simplify-sccs* are both false) and treat every allocation site of StringBuffer individually (merge-stringbuffer is false). Both SPARK and Paddle are running with on-the-fly call graph construction [Lhoták 2006]. We choose

<sup>&</sup>lt;sup>16</sup>http://www.sable.mcgill.ca/ashes/

ACM Transactions on Embedded Computing Systems, Vol. V, No. N, Article A, Publication date: January YYYY.

80 and 40 for the fractional parameters  $\delta_1$  and  $\delta_2$  (geom-frac-base is 40), which exhibit a good performance and precision trade-off. The classification of the code to user's and library's classes, required by constraints distillation technique, are judged from the package name. To be conservative, we only treat the package names with the prefixes **java**, **javax**, **sun** and **com.sun** as the library code. This configuration is conservative enough to allow our points-to engine to analyze all the user's code.

## 6.2. Performance

	Program	Preprocess (s)		Time (s)		Ν	lemory (MI	3)		
			SPARK	Geom-1	1-obj-W	SPARK	Geom-1	1-obj-W		
	jflex	0.3	45.6	73.6	2676.8	1123	4532	10621		
	soot	0.2	30.2	60.9	401.8	1023	2045	4571		
e	sableCC	0.3	49.5	83.1	3624.4	1677	5240	9392		
Suit	ps	0.2	34.5	88.0	484.0	1049	4006	4884		
2	antlr05	0.1	22.4	17.4	309.8	475	1293	3297		
dic.	bloat05	0.3	32.3	406.1	901.8	1127	5104	7688		
	pmd05	0.4	22.9	37.6	334.9	704	2115	3747		
	jedit1	0.9	91.8	495.7	-	2809	14479	-		
	megmek	0.7	103.2	435.8	-	2956	19418	-		
	antlr06	0.8	40.4	50.5	10814.7	925	2139	13593		
	bloat06	0.4	51.3	186.4	13115.5	1333	4283	19133		
te	jython	0.4	50.5	216.4	17699.2	1457	7465	25449		
Sui	luindex	0.7	38.3	47.8	8536.5	1175	2055	12598		
lle	pmd06	0.3	45.8	67.2	12117.4	1408	5060	14789		
lido	xalan	0.7	36.2	46.6	7223.3	951	1833	11808		
2	chart	0.9	161.3	541.0	32671.2	4969	21436	29393		
	eclipse	0.3	42.8	71.5	12560.0	1248	2494	19756		
	hsqldb	0.7	37.0	53.2	7197.2	1175	1917	11564		
e	batik	1.0	286.9	865.7	_	5761	25412	_		
Suit	sunflow	0.8	131.2	507.4	-	3786	14440	-		
e Se	tomcat	1.1	192.3	758.0	-	5391	21867	-		
arg	h2	0.6	82.2	196.8	-	2373	8390	-		
Г	jedit2	1.3	168.8	927.0	-	5953	27034	-		

Table V. Summary of the time and memory usage for all evaluated algorithms.

The time and memory usage for all the evaluated algorithms are collected in Table V, in which the subjects labeled "–" are those running out of memory. The running time for Geom excludes the time of SPARK but its memory usage includes the memory used by SPARK. From the table we see, even for the largest benchmark jedit2 with more than 26,000 methods, Geom can finish within 20 minutes (927s+169s). This is an important result because there is no reported context sensitive points-to analysis scales to this level. Statistically, Geom is on average <sup>17</sup> 7.1X and 81.9X faster than 1-obj-W in the small and middle suites, where the time usage for Geom counts for the time used by SPARK. In terms of the absolute running time, we claim our algorithm is more practical than the 1-obj-W implemented in Paddle considering that in reality,

ACM Transactions on Embedded Computing Systems, Vol. V, No. N, Article A, Publication date: January YYYY.

A:28

 $<sup>^{17}</sup>$ In this paper, average means the geometric mean.

the points-to analysis would run many times in the course of software testing and debugging, as well as in the development of the new research prototypes based on the points-to information.

Memory consumption is the major hindering factor of scaling the context sensitive points-to algorithms. On average, our algorithm Geom requires 1.9x and 4.5x less memory than 1-obj-W for the small and middle suites. The efficiency of our algorithm is significantly contributed by the constraints distillation and offline variable merging techniques. The column #Constraints shows 31.5% of the constraints extracted from SPARK are removed from the geometric points-to computation on average. And in our experience, removing these constraints retain 30% performance <sup>18</sup> while only using 1 second preprocessing time even for the largest benchmark. We believe more constraints can be removed in the case that the user performs a demand driven analysis to compute the refined points-to information for only a small fraction of pointers, in contrast to all the pointers in our experiment.

Although BDD is proven to be extremely memory efficient, its time efficiency is unsatisfiable. Many recent researchers reveal that, without BDD, we can do much better for the algorithms that produce less redundancy, especially in the case of heap cloning [Xu and Rountev 2008; Bravenboer and Smaragdakis 2009a; Hardekopf and Lin 2007a; Xiao and Zhang 2011]. In our design principle, time is a more critical resource than memory. We expect our algorithm can help those programmers, who require a much better memory disambiguation tool than SPARK, but they always work on large code base and have little patience on waiting for the points-to computation. The performance experiment has already successfully demonstrated that our algorithm is fast and scalable, in the next section, we will examine the precision quality of our algorithm especially the capability of reducing the aliased pointer pairs.

#### 6.3. Precision

**Average Points-to Set Size.** The average points-to set size is a widely used metric for precision assessment. Although the smaller the better, it cannot be pervasively used to predict the benefits to the clients. For example, it is not a good indicator for the clients call graph construction and casts safety analysis, because these clients desire less variance in types for the objects in the points-to set and, the smaller points-to set does not mean less types. However, it is in general a good precision indicator for the clients that need to traverse the points-to graph, such as side effect analysis, information flow analysis, and thread escape analysis, because smaller points-to set usually means less spurious information is propagated in these clients. Therefore, we give the metric here for whom may concern.

In this paper, we only consider the local pointers defined in the user's code and the instance field pointers with its base object allocated in the user's code. The points-to set for a pointer p is the merging of the points-to sets of p under all contexts. We list the results in Table VI. In summary, the numbers for the small suites show that SPARK has 2.3X and 3.3X larger points-to sets than Geom and 1-obj-W. 1-obj-W performs better than Geom especially on the case *antlr05*. Without this outlier case, the improvements over SPARK are 2.1X and 2.6X. On the middle suites, Geom and 1-obj-W have 2.6X and 2.5X smaller points-to sets than SPARK. These results show that Geom and 1-obj-W work closely well. And both of them produce much more precise points-to information than SPARK.

 $<sup>^{18}</sup>$  Our early paper reports 17% performance gain by using only the constraints distillation [Xiao and Zhang 2011].

ACM Transactions on Embedded Computing Systems, Vol. V, No. N, Article A, Publication date: January YYYY.

	Benchmark	Avg. F	oints-to '	Fuples		Safe Static Casts			
		SPARK	Geom	1-obj-W	Total	SPARK	Geom	1-obj-W	
	jflex	124.2	40.6	41.3	584	1	+0	+0	
	soot	74.5	41	35.5	1070	174	+9	+10	
e	sableCC	48.5	23.1	13.6	531	168	+1	+6	
Suit	$\mathbf{ps}$	135.3	107.7	148.8	676	17	+79	+79	
5	antlr05	54.2	13.2	3.5	80	16	+2	+3	
Mic	bloat05	160.5	112.3	69.1	1480	130	+7	+35	
F4	pmd05	35.9	10.2	6.1	295	12	+5	+5	
	jedit1	121.7	86.9	-	799	97	+13	-	
	megmek	136.1	102.9	-	1705	62	+7	-	
	antlr06	28.9	8.5	6.7	82	18	+2	+6	
	bloat06	172.6	66.9	72.5	1346	127	+4	+34	
te	jython	113.5	76.2	83.2	677	176	+4	+24	
Sui	luindex	18.5	4.9	6.8	63	12	+4	+10	
lle	pmd06	39.9	33.0	12.3	529	45	+6	+5	
lide	xalan	6.2	1.6	2.7	6	2	+0	+3	
2	chart	100.7	77.5	66.6	692	77	+10	+34	
	eclipse	59.9	17.7	19.1	312	76	+2	+13	
	hsqldb	8.8	1.7	3.1	12	2	+0	+3	
e	batik	243.3	174.4	-	885	129	+26	-	
Suit	sunflow	38.2	16.3	-	48	12	+5	-	
ge 6	tomcat	68.6.4	38.9	-	156	46	+6	-	
'ar	h2	200.9	66.7	-	968	68	+18	-	
	jedit2	180.9	111.5	-	972	149	+32	-	

Table VI. The average points-to tuples and the number of static casts proven to be safe.

**Static Casts safety.** The number of static casts that are proven to be safe is recently a widely cited metric [Sridharan and Bodík 2006; Xu and Rountev 2008; Smaragdakis et al. 2011]. This client is a good candidate for diagnosing the points-to information because the pointers involved in the casts would probably use the polymorphism feature of Java so that they always receive the objects from different calling contexts. Thus, the context sensitive points-to analysis can give full scope to its power.

We summarize the results in Table VI. Overall, 1-obj-W performs better than Geom. The reason is the static casts often happen in the places where the pointers read from a container (*i.e.* HashSet, ArrayList). Since the object sensitivity excels in analyzing the container data structures [Milanova et al. 2005], 1-obj-W is expected to perform better. Our observation is also consistent with Lhoták *et.al* [Lhoták and Hendren 2008], who also notice that the object context sensitive algorithms are better than the callsites based approaches in the static casts safety analysis. However, we will see in Section 6.4, as we iterate our algorithm one more time, the points-to results for static casts analysis can be greatly improved.

**Virtual Call Resolution.** Call graph construction is an important application of the points-to analysis. Since Java treats all the member functions as virtual functions, building the precise call graph is a challenging problem. We use the evaluated algorithms to build the context insensitive call graph (CICG). A virtual call is *solved* iff we have a unique callee for that callsite, where the callees for a callsite are decided by the

context insensitive points-to result by merging the points-to sets of the same pointer under different contexts. For all the unsolved callsites in ClCG, we also check the 1-CFA cloned call graph (1-CCG) to see how many of them can be solved under different calling contexts. The 1-CCG for Geom is constructed by naturally mapping the k-CFA abstraction to 1-CFA abstraction, while the 1-CCG for 1-obj-W is constructed by first collecting all the call edges that are associated to a particular object context, then we compute the set of object contexts for each call edge. With this information, we know for each function invoked by a particular call edge, which object contexts we should inspect.

Table VII. Virtual Call Resolution. CICG reports the number of resolved virtual callsites in the context insensitive call graph relative to to the SPARK column. 1-CCG reports the percentage of resolved virtual callsites in the *1*-CFA call graph. 39.1% (105) means 39.1% out of 105 callsites in the 1-CCG are solved.

	Benchmark	Total	CICG		1-CCG		
			SPARK	Geom	1-obj-W	Geom	1-obj-W
	jflex	2580	2561	+7	+7	39.1% (105)	1.5% (86)
	soot	5725	5384	+3	+2	8.9% (6701)	2.1% (6462)
e	sableCC	3949	3645	+15	+55	2.4% (461)	0.85% (351)
Suit	$\mathbf{ps}$	2338	2034	+3	+3	0.2% (3943)	0.0% (3941)
Lo Si	antlr05	4465	4070	+1	+2	2.4% (996)	0.0% (984)
Mic	bloat05	12479	11840	+22	+27	25.0% (8026)	3.0% (8100)
	pmd05	1985	1975	+0	+0	5.9% (17)	0.0% (17)
	jedit1	12089	11665	+35	-	4.6% (2258)	-
	megmek	31682	30667	+100	-	6.9% (4225)	-
	antlr06	4548	4158	+4	+0	3.1% (907)	0.1% (896)
	bloat06	12013	11410	+17	+44	24.9% (8272)	3.0% (8184)
te	jython	7006	6558	+25	+95	9.6% (2201)	3.3% (1505)
Sui	luindex	1195	1074	+57	+94	79.5% (122)	19.4% (31)
lle	pmd06	3632	3617	+3	+0	5.4% (37)	40.5% (53)
Lide	xalan	311	309	+2	+2	-	-
	chart	8639	8349	+106	+117	29.6% (1187)	12.9% (651)
	eclipse	3561	3448	+19	+21	14.5% (482)	4.8% (441)
	hsqldb	552	527	+17	+12	0.0% (26)	17.9% (41)
e	batik	7111	6642	+32	_	15.8% (2898)	-
Suit	sunflow	946	937	+15	-	0.0% (6)	-
e e	tomcat	2020	1946	+23	-	25.6% (496)	-
arg	h2	18795	16673	+55	-	5.5% (35832)	-
	jedit2	14880	14208	+40	-	13.4% (3873)	-

The result of virtual call resolution for the callsites in the user's code is collected in Table VII. Because Geom and 1-obj-W compute slightly different sets of reachable user' methods and 1-obj-W is more precise in the call graph construction <sup>19</sup>, we take the set obtained by 1-obj-W as the baseline. In the CICG construction, 1-obj-W is relatively better than our algorithm. The cases *sableCC* and *luindex* are notable, 1-obj-W works significantly better on them than Geom because they frequently employ the design patterns, specifically, the visitor pattern for *sableCC* and the strategy pattern for

<sup>&</sup>lt;sup>19</sup>Because 1-obj-W computes the call graph on-the-fly, but Geom uses a precomputed call graph by SPARK.

ACM Transactions on Embedded Computing Systems, Vol. V, No. N, Article A, Publication date: January YYYY.

*luindex*. Both patterns are data centric, *i.e.*, the way to modify the data is either decided by the input visitors or the overridden methods associated with the data object. Therefore, these programs are more naturally to be processed in the object sensitivity way. But still, we can run Geom one more time to significantly improve the CICG. The experimental result is given in Section 6.4.

It is interesting to find that our algorithm Geom shows strong potential in precisely constructing the 1-CCG. Taking the subject *jflex* as an example, the data 39.1% (105) means 39.1% of the 105 unsolved callsites <sup>20</sup> are solved. We cannot compare Geom and 1-obj-W directly because the constructed 1-CCG are different in the two algorithms <sup>21</sup>. Nevertheless, there are still quite a few callsites solved in the 1-CCG, especially for the cases *bloat06*, *luindex* and *chart*, *etc*. Therefore, Geom provides an opportunity to navigate the call graph traversal algorithms in a precise way.

**Alias Analysis.** A high quality aliased pointer expressions disambiguator is crucial to many programming tools. In Java, the field access expression p.f incurs the cross function information flow. Therefore, the less the alias-to pointers for the base pointer p, the more precise for tracking the information flow through heap variables. To evaluate the quality of the points-to analysis serving the alias queries, we find out all the base pointers p that appear in the field access expressions in the user's code, and then, we exhaustively enumerate two pointers p, q and intersect their points-to sets. The intersection does not account for the string constants and the heap sensitivity feature is used by Geom. Our earlier paper [Xiao and Zhang 2011] uses a different alias evaluation methodology, in which it enumerates all pairs of pointers accessed in the same function. Although this method is also used at other places [Das et al. 2001; Lattner et al. 2007], we do not follow the way because it is not meaningful to consider the pointers that cannot cause read or write conflicts to other pointers.

The alias analysis result is presented in Figure 10. We take the number of alias pairs in SPARK as the baseline, hence, the alias analysis quality for Geom and 1-obj-W is characterized as the percentage of alias pairs produced by SPARK. Geom and 1-obj-W respectively reduce 71.7% and 59.0% alias pairs made by SPARK (the out of memory cases for 1-obj-W are excluded), indicating that they both are significantly better than SPARK at the memory disambiguation and Geom is even better. Including all cases, Geom reduces 63.1% alias pairs than SPARK. This result is consistent with our earlier paper [Xiao and Zhang 2011]. In the cases where the number of contexts exceed  $2^{63}$  (overflowed), especially the subjects in the largest suite, the eliminated fake alias pairs are dramatically less than other programs. This is because the overflowed methods cannot enjoy the benefits of context sensitivity even with the *1*-CFA SCC modeling, which causes high damage to the precision.

The achievement of Geom in alias analysis, of course, attributes to the heap cloning and the *1*-CFA model for SCCs, although we use a small fractional parameter (*geom*-*frac-base* = 40) in the evaluation. Since our algorithm requires much less computing time and memory, the result reported in this paper is noteworthy and significant.

## 6.4. Iterative Analysis

Since our algorithm Geom reuses the call graph built by SPARK, the vast amount of spurious call edges significantly affect the subsequent context sensitive points-to computation. A simple idea is to update the call graph with the new points-to information computed by Geom and then, rerun Geom with the refined call graph. Following this

<sup>&</sup>lt;sup>20</sup>After the *1*-CFA cloning, the 12 unsolved callsites in the CICG are cloned to 105 callsites.

 $<sup>^{21}</sup>$  Therefore, the conclusion that Geom is more precise than 1-obj-W in our earlier paper [Xiao and Zhang 2011] is not valid.



Fig. 10. Alias analysis. Each bar stands for the percentage of alias pairs of that computed by SPARK.

	Benchmark	Time (s)	Avg. Poi	Avg. Points-to		Safe Casts		G
			1-obj-W	Geom	1-obj-W	Geom	1-obj-W	Geom
	antlr06	+29.0	-1.8	-0.7	+4	+1	-4	+0
	bloat06	+138.3	+5.6	-2.2	+30	+1	+27	+1
te	jython	+87.7	+7	-6.1	+20	+4	+70	+26
Sui	luindex	+26.2	+1.9	-0.5	+6	+0	+37	+40
lle	pmd06	+39.6	-20.7	-5.7	-1	+0	-3	+0
Lide	xalan	+33.2	+1.1	-0.4	+3	+0	+0	+0
	chart	+144.5	-10.9	-6.8	+24	+10	+11	+4
	eclipse	+43.5	+1.4	-2.2	+11	+1	+2	+0
	hsqldb	+28.3	+1.4	-0.4	+3	+0	-5	+0

Table VIII. Performance and precision data for running Geom2.

process, we can run Geom again and again until the call graph cannot be refined. We call this procedure *iterative analysis* and expect the precision to be greatly improved. To prove our claim, we collect the performance and precision data of the middle suites in Table VIII and Table IX by running Geom twice and thrice. We refer to the two iterative analyses as Geom2 and Geom3. Interesting readers could use the soot phase option *geom-runs* to set how many runs are performed by the geometric analysis.

The data of Geom2 and Geom3 are given in relative to the corresponding values of Geom. For example, the value "+29.0" in the program *antlr06* means that Geom2 needs another 29 seconds to finish the analysis. The results for 1-obj-W are also given in relative form to help the readers observe the power of iterative analysis intuitively. It is interesting that the running time for Geom2 and Geom3 is not 2X and 3X to that of

	Benchmark	Time (s)	Avg. Poi	Avg. Points-to		Safe Casts		G
			1-obj-W	Geom	1-obj-W	Geom	1-obj-W	Geom
	antlr06	+61.7	-1.8	-0.7	+4	+1	-4	+0
	bloat06	+272.9	+5.6	-2.3	+30	+1	+27	+1
te	jython	+205.9	+7	-6.1	+20	+6	+70	+30
Sui	luindex	+55.2	+1.9	-0.6	+6	+1	+37	+40
lle	pmd06	+70.3	-20.7	-5.7	-1	+0	-3	+0
Lide	xalan	+47.6	+1.1	-0.4	+3	+0	+0	+0
2	chart	+376.2	-10.9	-7.8	+24	+10	+18	+11
	eclipse	+82.7	+1.4	-2.2	+11	+1	+2	+0
	hsqldb	+53.1	+1.4	-0.4	+3	+0	-5	+0

Table IX. Performance and precision data for running Geom3.

Geom. I(n fact, the execution times for Geom2 and Geom3 are only on average 1.6X and 2.1X to Geom. This is because each iteration would significantly reduce the call graph size. Also, from the second iteration on, we use the geometric points-to result computed so far to boot the constraints distillation. By these means, the analyzed constraints in the second and third runs are dramatically reduced and, the computing time for subsequent iterations becomes less and less  $2^{22}$ .

We also collect the call graphs and the constraints information for the programs in the middle suites in Table X to highlight the improvements by the iterative analysis. We can draw three conclusions from the table. First, the usefulness of iterative analysis is prominent since both Geom2 and Geom3 are much better than Geom in all three metrics. Second, the precision of Geom2 and Geom3 is almost identical. This result suggests that *running the geometric analysis twice in practice is a good choice*. Third, 1-obj-W is better in the call graph construction. This is expected because 1-obj-W constructs the call graph on-the-fly, which is realized as a better way than using a precomputed call graph [Lhoták 2002]. However, exactly because of using the precomputed call graph, we can get rid of the restriction that many contexts are unknown at the beginning and encode the contexts in a concise way, and finally, promote the points-to analysis to a very large scale.

# 7. RELATED WORK

Points-to analysis is a well studied subject of a large body of work. We have given a detailed classification of the points-to literature in Section 1.2. In this section, we compare our work to the most related and recent algorithms, all of which are in the category of designing or engineering an extremely fast yet still precise points-to analysis. Our comparison focuses on the implementation issues, and, we explain the advances of our geometric points-to analysis over theirs.

One of the pioneer that recognizes the importance of data representation, **bddbddb**, is a highly flexible engine designed by Whaley *et al.*, aimed for prototyping a range of points-to based algorithms [Whaley and Lam 2004]. The points-to analysis shown in bddbddb is the first scalable full context sensitive analysis. Compared to our work, bddbddb lacks the support for heap cloning and *1*-CFA model of SCCs. Therefore, as pointed out by Lhoták [Lhoták and Hendren 2008], the precision of bddbddb is only comparable to the 1-callsite-sensitive analysis, which is both less precise and less efficient than the 1-object-sensitive analysis used in our evaluation. We cannot di-

 $<sup>^{22}</sup>$ Timing may be affected by garbage collection significantly, thus, some programs such as *jython* need more time in the third iteration than the second.

		SPARK	1-obj-W	Geom	Geom2	Geom3
r	#total methods	9831	7266	8749	8397	8395
untl	#user methods	810	754	775	773	773
6	#constraints	_	_	68.2%	55.9%	54.0%
,t	#total methods	11745	9133	10696	10347	10345
loa	#user methods	2603	2487	2541	2540	2540
	#constraints	-	-	73.2%	62.2%	60.9%
u	#total methods	12729	8785	10486	10163	10138
the	#user methods	3470	2072	2244	2108	2089
ví	#constraints	-	-	76.0%	48.4%	45.0%
ex	#total methods	9593	6874	8422	8003	8001
ind	#user methods	541	354	422	378	378
lu	#constraints	-	-	67.3%	53.5%	51.0%
	#total methods	11103	8302	9886	9540	9538
bme	#user methods	2008	1750	1793	1791	1791
	#constraints	-	-	69.2%	56.0%	54.5%
u	#total methods	9106	6579	8052	7685	7683
ala	#user methods	810	57	57	57	57
×	#constraints	-	-	66.5%	53.6%	51.1%
ц	#total methods	25083	11733	20984	19339	19192
thai	#user methods	3168	2390	2599	2576	2576
5	#constraints	-	-	66.6%	50.1%	45.8%
se	#total methods	10372	7707	9245	8801	8798
clip	#user methods	1125	998	1031	1027	1027
ē	#constraints	-	-	69.4%	55.8%	53.7%
db	#total methods	9104	6559	8036	7666	7664
sqle	#user methods	58	54	54	54	54
hs	#constraints	-	_	66.8%	53.6%	51.1%

Table X. Statistics for call graph and constraints.

rectly compare our implementation with bddbddb, because their compiler models the program in a different way from Soot [Bravenboer and Smaragdakis 2009a]. But as shown by Lhoták [Lhoták and Hendren 2008], the Paddle's version of Whaley's algorithm is slower than the 1-object-sensitive analysis, which is an evidence to claim that our performance superiority over bddbddb.

The EPA algorithm is the first, in our knowledge, to realize the limitation of improving the efficiency of a precise points-to algorithm with BDD [Xu and Rountev 2008]. Instead, it presents an interesting way to compress the points-to information by merging the equivalent contexts that yield a set of points-to tuples in the same structure. Our geometric encoding can be seen as a simpler and more compact interpretation of their core idea with the extension to handling the globals more precisely. However, the EPA algorithm has a sophisticated implementation that we did not manage to successfully port it to our experiments, thus, we cannot compare to EPA directly in this paper. However, as a functional approach, EPA needs to perform additional procedures to compute the escaped objects, instantiate and merge the symbolic objects, and remove the context information for the objects pointed to by the global variables. From their comparison to the BDD based 1-object-sensitive analysis and the fact that BDD based 1-object-sensitive analysis is orders of magnitude slower than the worklist version [Xiao and Zhang 2011], we conjecture that EPA is hard to beat our algorithm that

is not bundled with extra procedures. Of course, we will try in future to make a fair comparison to EPA.

Doop is the most recent one to reveal that BDD is not a proper data representation for the points-to analysis producing less redundancy [Bravenboer and Smaragdakis 2009a]. Doop is fully declarative points-to framework with rich features. including the declarative on-the-fly call graph construction and exception handling. Similar to our work, Doop's high performance is also obtained from non-BDD based points-to representation and constraints evaluation. But Doop does not develop any compressed representation so that its points-to and pointer assignments information is explicitly stored without compaction. Since reproducing their experiments is almost impossible due to their use of a highly optimized Datalog engine, we doubt that whether or not the Doop's approach is a practical solution to be widely adopted. Our geometric points-to analysis is much simpler to be re-implemented, since no particular software is used in our design. As our submitted code has been integrated into Soot, we believe our solution will be a easier way to help the researchers experience the power of a fast and precise points-to analysis.

### 8. CONCLUSION AND FUTURE WORK

In this paper, we present an efficient and precise context sensitive points-to analysis with heap cloning, based on our simple and compact geometric encoding. Our new algorithm has excellent performance, which is on average 81.9X faster than the worklist based 1-object-sensitive analysis implemented in Paddle. Meanwhile, with extensive experiments we show that their precision is close. Our geometric analysis implementation has been accepted by the Soot maintenance team and it is available since version 2.5. Our future work will consider how to incorporate the on-the-fly call graph construction with geometric encoding. We have faith that, all the techniques (geometric encoding, constraints distillation and *1*-CFA model for SCCs) proposed in this paper make important contributions to engineer a fast and precise context sensitive points-to analysis for the large scale software.

## Acknowledgements

Our premilinary version of this paper is carefully commented by the ISSTA 2011 reviewers, we thank again for their insightful suggestions. We also thank the users of our analysis to report bugs and give suggestions for the improvement after our implementation shipped with Soot.

## REFERENCES

- ANDERSEN, L. O. 1994. Program analysis and specialization for the c programming language. Ph.D. thesis.
- BERNDL, M., LHOTÁK, O., QIAN, F., HENDREN, L., AND UMANEE, N. 2003. Points-to analysis using bdds. In Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation. ACM.
- BODDEN, E., SEWE, A., SINSCHEK, J., OUESLATI, H., AND MEZINI, M. 2011. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*. ICSE '11.
- BRAVENBOER, M. AND SMARAGDAKIS, Y. 2009a. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA '09*. ACM.
- BRAVENBOER, M. AND SMARAGDAKIS, Y. 2009b. Strictly declarative specification of sophisticated points-to analyses. In Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications. OOPSLA '09.
- CHATTERJEE, R., RYDER, B. G., AND LANDI, W. A. 1999. Relevant context inference. In Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '99.

ACM Transactions on Embedded Computing Systems, Vol. V, No. N, Article A, Publication date: January YYYY.

A:36

- CHENG, B.-C. AND HWU, W.-M. W. 2000. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *Proceedings of the ACM SIGPLAN 2000 conference on Pro*gramming language design and implementation. PLDI '00.
- COUSOT, P. AND COUSOT, R. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN* symposium on *Principles of programming languages*. POPL '77.
- CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. 13.
- DAS, M., LIBLIT, B., FÄHNDRICH, M., AND REHOF, J. 2001. Estimating the impact of scalable pointer analysis on optimization. In SAS '01. Springer.
- EMAMI, M., GHIYA, R., AND HENDREN, L. J. 1994. 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*. ACM.
- FÄHNDRICH, M., FOSTER, J. S., SU, Z., AND AIKEN, A. 1998. Partial online cycle elimination in inclusion constraint graphs. In Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation. PLDI '98.
- FÄHNDRICH, M., REHOF, J., AND DAS, M. 2000. Scalable context-sensitive flow analysis using instantiation constraints. In Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation.
- HARDEKOPF, B. AND LIN, C. 2007a. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the ACM SIGPLAN 2007 conference on Programming language design and implementation*. ACM.
- HARDEKOPF, B. AND LIN, C. 2007b. Exploiting pointer and location equivalence to optimize pointer analysis. In In International Static Analysis Symposium (SAS. 265–280.
- HASTI, R. AND HORWITZ, S. 1998. Using static single assignment form to improve flow-insensitive pointer analysis. In Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation. PLDI '98.
- HEINTZE, N. AND TARDIEU, O. 2001. Demand-driven pointer analysis. In Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation. PLDI '01.
- HIND, M., BURKE, M., CARINI, P., AND CHOI, J.-D. 1999. Interprocedural pointer alias analysis. ACM Trans. Program. Lang. Syst. 21.
- LANDI, W. AND RYDER, B. G. 1992. A safe approximate algorithm for interprocedural aliasing. In Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation. PLDI '92.
- LATTNER, C., LENHARTH, A., AND ADVE, V. 2007. Making context-sensitive points-to analysis with heap cloning practical for the real world. *PLDI* '07.
- LHOTÁK, O. 2002. Spark: A flexible points-to analysis framework for Java. M.S. thesis, McGill University.
- LHOTÁK, O. 2006. Program analysis using binary decision diagrams. Ph.D. thesis, Montreal, Canada.
- LHOTÁK, O. AND HENDREN, L. 2008. Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. ACM TOSEM.
- LHOTÁK, O. AND LAURIE, H. 2003. Scaling java points-to analysis using spark. CC '03 2622.
- LIANG, D. AND HARROLD, M. J. 2001. Efficient computation of parameterized pointer information for interprocedural analyses. In *Proceedings of the 8th International Symposium on Static Analysis*. SAS '01.
- MICHA, S. AND PNUELI, A. 1981. Two approaches to interprocedural data flow analysis. In S. Muchnick and N.D Jones, editors, Program Flow Analysis: Theory and Applications, 189–234.
- MILANOVA, A., ROUNTEV, A., AND RYDER, B. G. 2005. Parameterized object sensitivity for points-to analysis for java. ACM TOSEM.
- NYSTROM, E. M., SEOK KIM, H., AND MEI W. HWU, W. 2004. Importance of heap specialization in pointer analysis. In *PASTE '04*. ACM.
- PEARCE, D. J., KELLY, P. H. J., AND HANKIN, C. 2003. Online cycle detection and difference propagation for pointer analysis. In *IEEE SCAM* '03.
- ROUNTEV, A. AND CHANDRA, S. 2000. Off-line variable substitution for scaling points-to analysis. In Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation. PLDI '00.
- ROUNTEV, A., MILANOVA, A., AND RYDER, B. G. 2001. Points-to analysis for java using annotated constraints. In Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. OOPSLA '01.

- ROUNTEV, A. AND RYDER, B. G. 2001. Points-to and side-effect analyses for programs built with precompiled libraries. In CC '01.
- SHIVERS, O. G. 1991. Control-flow analysis of higher-order languages of taming lambda. Ph.D. thesis, Carnegie Mellon University.
- SMARAGDAKIS, Y., BRAVENBOER, M., AND LHOTÁK, O. 2011. Pick your contexts well: Understanding object-sensitivity. In Proceedings of the 38th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM.
- SRIDHARAN, M. AND BODÍK, R. 2006. Refinement-based context-sensitive points-to analysis for java. *PLDI* '06.
- SRIDHARAN, M. AND FINK, S. J. 2009. The complexity of andersen's analysis in practice. In *Proceedings of* the 16th International Symposium on Static Analysis. SAS '09.
- VALLEE-RAI, R., GAGNON, E., HENDREN, L., LAM, P., POMINVILLE, P., AND SUNDARESAN, V. 2000. Optimizing java bytecode using the soot framework: Is it feasible? In CC '00.
- WHALEY, J. 2007. Context-sensitive pointer analysis using binary decision diagrams. Ph.D. thesis, Stanford University.
- WHALEY, J. AND LAM, M. S. 2004. 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. ACM.
- WILSON, R. P. AND LAM, M. S. 1995. Efficient context sensitive pointer analysis for c programs. In Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation.
- XIAO, X. AND ZHANG, C. 2011. Geometric encoding: forging the high performance context sensitive points-to analysis for java. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ISSTA '11.
- XU, G. AND ROUNTEV, A. 2008. Merging equivalent contexts for scalable heap-cloning-based contextsensitive points-to analysis. In *ISSTA* '08. ACM.
- YAN, D., XU, G., AND ROUNTEV, A. 2011. Demand-driven context-sensitive alias analysis for java. In Proceedings of the 2011 International Symposium on Software Testing and Analysis. ISSTA '11.
- YU, H., XUE, J., HUO, W., FENG, X., AND ZHANG, Z. 2010. Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. CGO '10.
- ZHU, J. AND CALMAN, S. 2004. Symbolic pointer analysis revisited. In Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation. PLDI '04.