Geometric Encoding: Forging the High Performance Context Sensitive Points-to Analysis for Java

Xiao Xiao Charles Zhang
Computer Science and Engineering Department
The Hong Kong University of Science and Technology
{richardxx, charlesz}@cse.ust.hk

ABSTRACT

Context sensitive points-to analysis suffers from the scalability problem. We present the geometric encoding to capture the redundancy in the points-to analysis. Compared to BDD and EPA, the state of the art, the geometric encoding is much more efficient in processing the encoded facts, especially for the high-order context sensitivity with the heap cloning. We also developed two precision preserving techniques, constraints distillation and *1*-CFA SCC modeling, to further improve the efficiency, in addition to the precision performance trade-off scheme. We evaluate our points-to algorithm with two variants of the geometric encoding, Geom and HeapIns, on 15 widely cited Java benchmarks. The evaluation shows that the Geom based algorithm is 11x and 68x faster than the worklist/BDD based 1-object-sensitive analysis in Paddle, and the speedup steeply goes up to 24x and 111x, if the HeapIns algorithm is used. Meanwhile, being very efficient in time, the precision is still equal to and sometime better than the 1-object-sensitive analysis.

Categories and Subject Descriptors

F.3.2 [**Semantics of Programming Languages**]: Program analysis

General Terms

Algorithms, Languages, Performance

Keywords

Geometric, Encoding, Points-to, Context, Sensitive

1. INTRODUCTION

Points-to analysis determines, given a pointer p, the set of heap allocation sites that p may point to. The recent studies [18, 10, 16, 5, 20] focus on points-to analysis with context sensitivity to improve the precision. *Context* is a static abstraction for distinguishing the different runtime invocations of the same function, typically represented by unique *call-strings* [6]. The context-sensitive points-to analysis is commonly reasoned under the k-CFA ($k \geq 0$) analysis framework [14], where k is the length of the path ascending from any given 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 [11], 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 considered as the most non-scalable case[1], and the corresponding technique can be easily translated to other cases with smaller k.

Prior Work. The earlier work [3, 19] has developed nearly all of the key treatments for handling context sensitivity. However, these approaches are inefficient because the points-to information is not efficiently stored and the constraints are evaluated individually for each context.

Whaley *et.al.* [18] presented the first practical solution that uses the *cloning*-based analysis, to achieve the full context sensitivity on very large realistic Java programs. They 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 [17], 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 more than one context-sensitive variables simultaneously, as in the case of heap cloning [7]. Besides, Hardekopf *et.al.* [4] reported that the use of BDD in Anderson's algorithm for C is around 2x slower than the sparse bitmap encoding. Bravenboer *et.al.* [1] even achieved 15x speedup on average with the Doop engine, compared to the BDD based context sensitive analyses in Paddle [6]. Based on these facts, BDD is limited in achieving our design objective of offering a practical analysis for both pointer and heap sensitivity.

An alternative approach for achieving full context sensitivity is the use of procedure summary [9, 5, 20], which analyzes every function separately with parameters. This flexibility requires an additional effort for the parameters instantiation and the computation of the escaped information, a tradeoff not necessary for the whole program analysis. Among all the algorithms, Lattner *et.al.*'s work [5] is quite scalable due to the use of Steensgard's unification for pointer assignments. Xu *et.al.*'s work [20] also employs the

[1]It is also in theory the most precise case for call-string based context model, if the SCCs are treated well.

Permission to make digital or hard copies of all or part 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 bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISSTA '11, July 17-21, 2011, Toronto, ON, Canada
Copyright 2011 ACM 978-1-4503-0562-4/11/07 ...$10.00.

unification, *e.g.*, for the pointers appear at the call sites. Additionally, Xu *et.al.* designed a compression technique for eliminating redundancy in the traditional points-to representation, to minimize the time and memory consumption. They find that many points-to tuples share the same calling context prefixes and, hence by removing them, two points-to tuples can become identical[2]. 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 explored (Section 2.3). Moreover, despite that the performance of EPA is significantly better than BDD based Paddle [6], its absolute running time is still unsatisfactory for the practical use. For instance, the medium sized program jflex runs 535 seconds [20], which is 50x slower than ours.

Our Contribution. To address the shortcomings of both the BDD and the summary based algorithms, we have designed an *non-BDD* and the *cloning* based context sensitive points-to analysis. Our algorithm is bootstrapped by Anderson's analysis, which helps us build the initial call graph. The main algorithm makes use of three techniques. The first is a new context encoding scheme called the *Geometric Encoding*, which encodes the points-to and the pointer assignment relations as regular geometric figures. Compared to EPA, the geometric encoding is simpler to implement and has higher compression. It is also flexible for the performance and the precision tuning. The second is a preprocessing step called constraints distillation. In Java, most of the constraints extracted for points-to analysis come from the Java library. However, not all the library code affects the points-to information in user's code. By wiping out the inconsequential library code prior to the constraints evaluation, we improve the performance without any precision loss. The third is a *1*-CFA model for handling the recursive calls. As shown by Lhoták [7], 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 program always has big SCCs (Section 4). 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.

Organization. The paper is organized as follows. Section 2 provides the details of geometric encoding and its inference rules, and the *1*-CFA model for SCCs. Section 3 introduces our points-to algorithm along with the constraints distillation technique. Finally, we exhibit our experimental results in Section 4, discuss the related work in Section 5 and conclude our paper in Section 6.

2. GEOMETRIC ENCODING

Consider performing Anderson's analysis on the sample program in Figure 1(a) under the full context sensitivity model for both the pointer and heap variables. As in Whaley's approach [18], we first locally number the contexts of each function by the integers 1, 2 . . . N, where N is the number of acyclic paths from main to that function on the SCC-condensed call graph shown in Figure 1(b). Our aim is to conclude that each version of pointer $list2$ under the contexts 1, 2, 3 of the function work points to all versions of o^{35} in the contexts 4, 5, 6 of the function addList.

To achieve this, the conventional analysis requires to evaluate the constraint $p = x$ (x returned by addList at Line 30) six times, $gList = p$ three times, and $list2 = gList$ nine times[3]. However, with the following two observations, this computation can be dramatically simplified. First, the return of addList induces six copies of *independent* assignments $p_i = x_i$, $\forall i \in [1, 7)$, which means x **under the** i^{th} **context only assigns to** p **under the** i^{th} **context**. Such *1-to-1 mapping* can be geometrically interpreted as a line segment using the context numbers of p and q as the coordinates. Second, consider the returns from createNewList to init and then to work, inducing the assignments $gList = p$ (Line 21) and $list2 = gList$ (Line 10). Because $gList$ is a global with one context only, thereby **any versions of** p **can be assigned to any versions of** $list2$, written as $list2_j = p_i$, $\forall i \in [4, 7)$, $j \in [1, 4)$. This *many-to-many mapping* between p and $list2$ can be interpreted as a bounded rectangle using the context numbers of p and $list2$ as the coordinates.

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. In our example, since both relations, x points-to o^{35} and x assigns-to p, form 1-to-1 mappings, the fusion of the these two mappings immediately yields the fact that p_i points to o^{35}_i for $i \in [1, 6]$ (o^{35}_i stands for the object o^{35} created under the ith context). Likewise, the many-to-many mapping between variables p and $list2$ described earlier yields the fact that each versions of $list2$ under the context 1, 2, 3 points to all versions of o^{35} created under the contexts 4, 5, 6. Computing these points-to facts geometrically only evaluates the corresponding constraints *once* independent of the number of contexts.

The discussion above refers to all the essential steps of applying the geometric encoding for points-to analysis: Number the contexts, encode the initial relations, reason the points-to relations via the encoded assignments. Next, we describe these steps in detail.

2.1 Contexts Naming

We first construct the context insensitive call graph G_i, then add an artificial function, **Super_Main**, to G_i as the entry point. **Super_Main** calls the **main** function, the static class initializers, and other possible entry points for Java programs. Since G_i may not be cycle free, we collapse all the SCCs in G_i to build the *reduced call graph* G_c, which is a condensed DAG of G_i. We then temporarily modify the *Non-SCC call edge* $X \to Y$ to $rep[X] \to rep[Y]$ if $rep[X] \neq rep[Y]$, where $rep[x]$ returns the identifier of the SCC that x belongs to. Correspondingly, if $rep[X] = rep[Y]$, the edge $X \to Y$ is called the *SCC call edge.*

Next, we apply Algorithm 1 on G_c to locally number the contexts for each function X by the integers 1, 2, . . . , N, where N is called the *context size*, denoted by $csize[X]$, standing for the number of acyclic call paths on G_c from **Super_Main** to X. The term *context bar* describes all the contexts of X in an interval form $[1, csize[X]]$. The *context mapping* between two functions for a callsite $X \to Y$ is captured by $callmap[X \to Y] = offset$, which maps any context of X to a context of Y by adding *offset* to that context of

[2]For example, (p, ζ_1, o, ζ_2) denotes that the pointer p under context ζ_1 points to the object o allocated under context ζ_2.

[3]Because $list2 = gList$ has three context sensitive versions, and for each context, $gList$ points to all the three versions of o^{35} under the contexts 4, 5, 6.

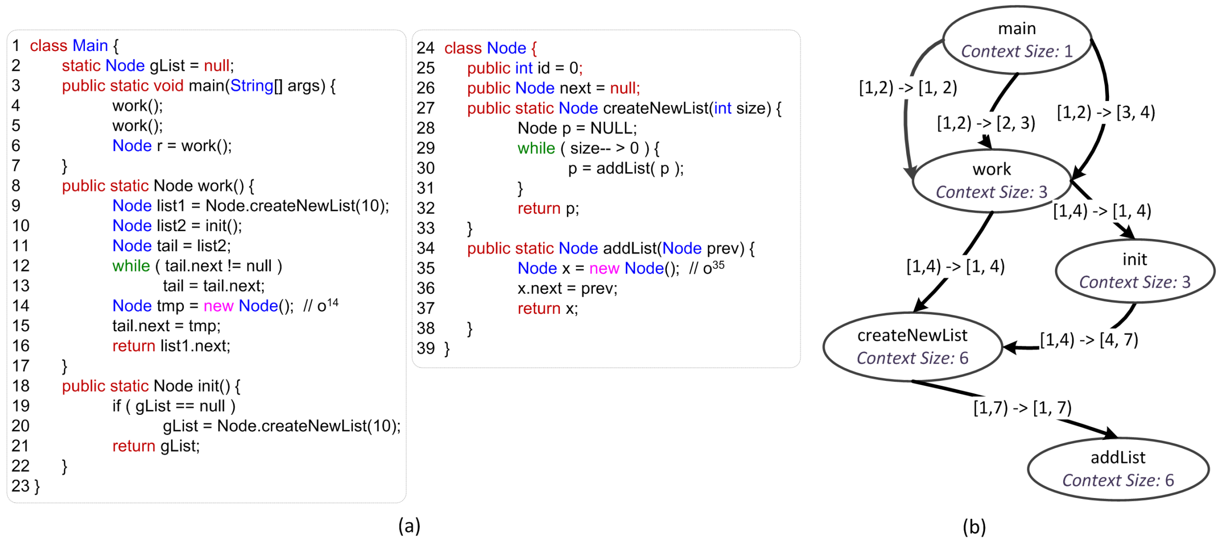


Figure 1: Sample source code and its call graph. This example is used throughout the paper.

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 1 *Context_Naming*(G_c: Reduced Call Graph)

0. $\forall$ function X, $csize[X] = 0$
1. $csize[Super_Main] = 1$
2. We visit the nodes of G_c in topological order
3. **for each** node X in G_c
4.   **for each** non-SCC call edge $X \to Y$
5. $callmap[X \to Y] = csize[Y] + 1$
6. $csize[Y] = csize[Y] + csize[X]$
7. **for each** none representative node X of SCC
8. $csize[X] = csize[rep[X]]$
9. **for each** SCC call edge $X \to Y$
10. $callmap[X \to Y] = 1$

Algorithm 1 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 5). We then increase $csize[Y]$ by $csize[X]$, which determines the number of contexts mapped from X to Y (Line 6). This way for building the context mapping arranges the contexts induced by X consecutively on Y, which is crucial for us to use the simple regular shapes for encoding the mapping relations. Moreover, we handle the SCC call edges in the *0*-CFA form (Line 7-10). 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 1 to handle the SCC more precisely in Section 2.5.

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

2.2 Geometric Encoding System

Same to other encoding methodologies, we encode the program facts extracted from the source code before points-to analysis. We call these encoded facts *initial encoding*.

2.2.1 Constructing the Initial Encoding

Pointer Assignments. The initial encoding is built on the *canonical form* of the program, in which it has only three types of pointer assignments: local-to-local, local-to-global and global-to-local. The assignment between two globals can be reduced to an assignment to a local, followed by an assignment to another global.

We represent the local-to-local assignment such as q = p in function X by $p \rightsquigarrow q$. The assignment has the semantics that p under the ith context only assigns to q under the ith context, written as $p_i \rightsquigarrow q_i, \forall i \in [1, csize[X]]$. 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 $q_1 = p_1$, $q_2 = p_2$, $\ldots$, $q_{csize[X]} = p_{csize[X]}$. This expression faithfully captures the *1-to-1 mapping* between p and q.

The function *callmap* computed by Algorithm 1 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 $p \rightsquigarrow q$ is expressed as $(p, q, 1, K, csize[X])$. Correspondingly, in the case of function return, the context sensitive version of $p \rightsquigarrow q$ is encoded as $(p, q, K, 1, csize[X])$.

The assignments with globals are more involved 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-1 mapping* and its geometric interpretation is a horizontal segment. Conversely, the assignment $p = g$ is a *1-to-many mapping* that represents a vertical segment. More sophisticated 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-many mapping*, figured as a rectangle.

We now define our symbols formally:

Definition 1. *The geometric encoding is a five or six tu-*

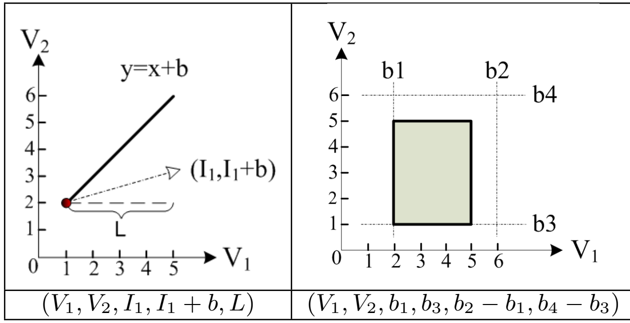


Figure 2: Illustration of the encoding tuples.

ple, in the form $(V_1, V_2, x_1, y_1, L_1, L_2)$ *(no* L_2 *term if it is a 5-tuple), abbreviated as* E_{V_1/V_2}*. The first two terms* (V_1, V_2) *form an* **interpreter tuple***, stating the labels of the X and Y axes. The following 4-tuple* (x_1, y_1, L_1, L_2) *is the* **geometric extension***, which describes the mapping relation between* V_1 *and* V_2*.*

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 Figure 2:

Definition 2. *The geometric extension describes a 2D geometric figure, which states the mapping relation between two variables. The two types of the geometric figures are:*

The diagonal segment $y = x + b$*. It indicates the 1-to-1 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.*

Points-to. Encoding the points-to facts is exactly the same as encoding the assignments. We first name the new expression (*e.g.* new Object) by an unique name o, then we treat o as a local if o is created in a function, or a global if created in a static initializer for a class. Next, we encode an allocation p = o in the same way as encoding an assignment (assume both p and o are local to function X), and obtain the outcome $(p, o, 1, 1, csize[X])$, in which p is written as the first term.

Pointer Dereference Assignments. The pointer dereference or the complex constraints include both the *load* and the *store* constraints that access the instance fields. For example, $q.f$ accesses the f field of the object pointed to by q. The geometric extension of the complex constraints $q.f = p$ (or $q = p.f$) is exactly the geometric extension of its corresponding simple assignment $q = p$, because it is sufficient for us to *instantiate* $q.f$ (or $p.f$) with the points-to result of q (or p) by the mapping relation between p and q.

Example. The initial encoding of our running example is included in Figure 3(a), where the geometric extensions are drawn on the arrows. Taking the assignment $gList = p$ as

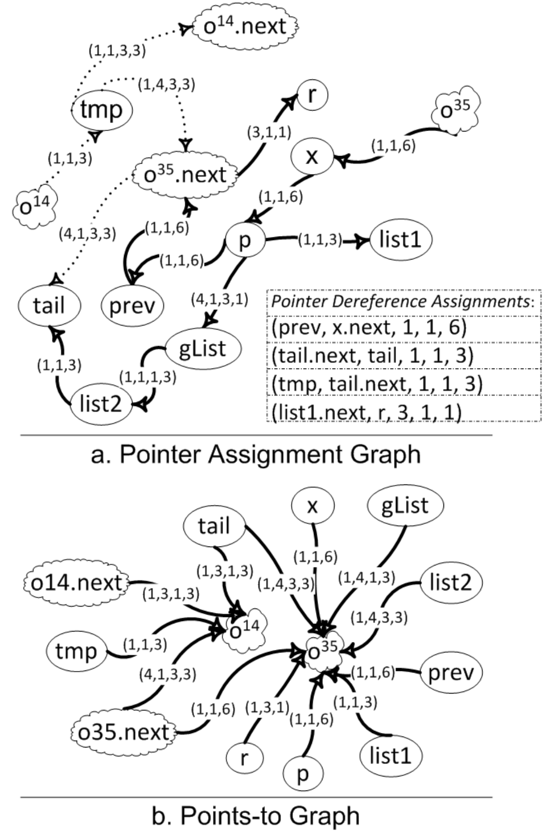


Figure 3: The graphic form of the final assignments and the points-to solution of our sample code. We render the paths that the objects o^{14} **and** o^{35} **can go through by dotted and solid arrowed lines.**

an example, we encode it as $(p, gList, \mathit{4}, \mathbf{1}, \mathit{3}, \mathbf{1})$, meaning all versions of p under the contexts $[\mathit{4}, \mathit{4} + \mathit{3})$ are assigned to the single version $([\mathbf{1}, \mathbf{1} + \mathbf{1}))$ of $gList$. During the analysis, the complex constraints instantiation adds the edges connected to $o^{35}.next$ and $o^{14}.next$.

2.2.2 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 $(p, o^{35}, 1, 1, 6)$ and $(p, gList, 4, 1, 3, 1)$, the fusion result is $(gList, o^{35}, 1, 4, 1, 3)$. This is obtained in four steps. First, we extract the context ranges of pointer p[4] in the points-to and assigns-to figures, which are $[1, 7)$ and $[4, 7)$. Second, we intersect the ranges and obtain the common interval $[4, 7)$. We call this step *clipping*. Third, we compute the intervals of $gList$ and o respecting to the interval $[4, 7)$ of p. In our case, they are $[1, 2)$ and $[4, 7)$. Finally, we compute the mapping relation between q and o, and we call this step *expanding*. Since all p_i $(i \in [4, 7))$ assign to the single copy of $gList$, it is a many-many mapping hence encoded by a rectangle: $(gList, o^{35}, 1, 4, 1, 3)$.

Instantiating the complex constraint can be performed in the same way as evaluating the pointer assignment, repre-

[4]Since p appears in both tuples, it is called the *agent pointer*.

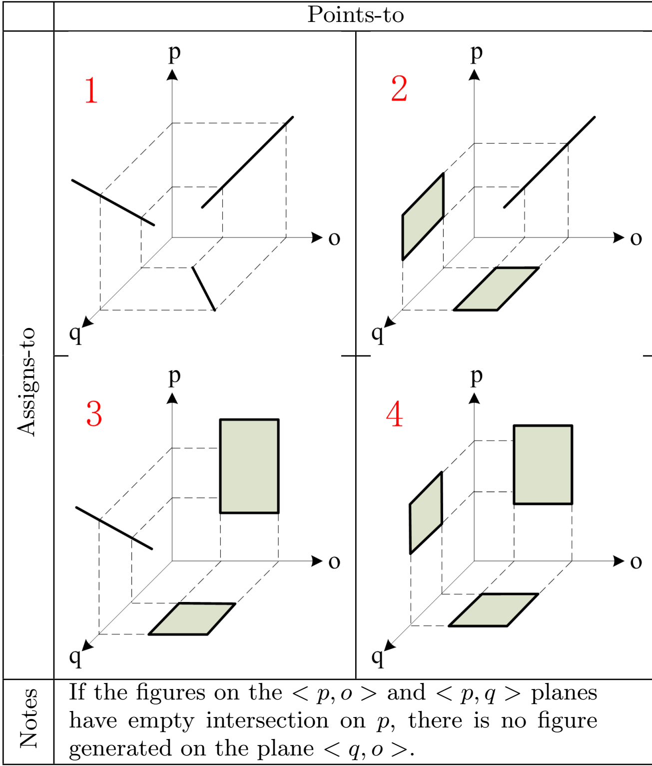


Figure 4: Assignment Inferring Rules. For each picture, the figures on the planes $< p, o >$ and $< p, q >$ are given as input, describing the p points-to o relation and the p assigns-to q relation. The generation steps of the figure on the plane $< q, o >$ are implicitly stated by the dashed lines, which stand for the clipping and expanding operations.

sented by the operator $\bullet$. For example, we instantiate the constraint $(tmp, tail.next, 1, 1, 3)$ against the points-to fact $(tail, o^{35}, 1, 4, 3, 3)$. We also need four steps of calculation discussed above, but this time the agent pointer is $tail$ and its intersected interval is $[1, 4)$ in the clipping step. After mapping the intersection interval to the variables tmp and o^{35}, we obtain $(tmp, o^{35}.next, 1, 4, 3, 3)$.

We visualize the inference rules for the pointer assignments in Figure 4[5], and the application of these rules to our discussed instances above are given in Figure 5. Each inference rule contains three mutually perpendicular planes, and the input geometric figures are given in the planes $< p, o >$ and $< p, q >$. We deduce the figure on the plane $< q, o >$ by the clipping and expanding steps explained before, which are rendered by the dashed lines. From Figure 4, we conclude that the diagonal segment and the rectangle $w.r.t$ the binary operations $\circ$ and $\bullet$, form two magma algebraic structures respectively. This result is important in two folds. First, it shows that the inherit difficulty of the geometric encoding is quite low, because only two simple geometric figures are involved. Second, the soundness of the inference rules can be verified easily by manual calculation, thus, we omit the proof for the following lemma:

[5]Because the complex constraints instantiation are very similar, we thereby elude the details.

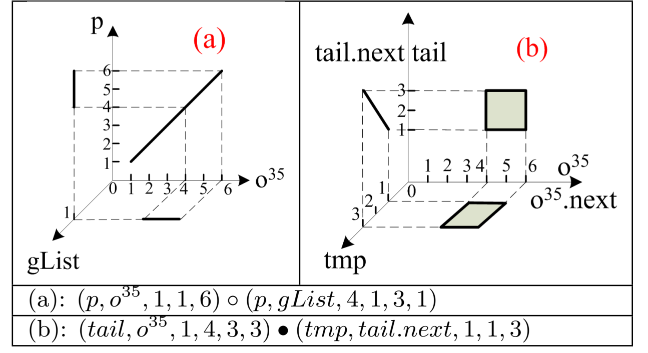


Figure 5: Exemplify the usage of the inference rules.

LEMMA 2.1. *The inference rules for $\circ$ and $\bullet$ are sound.*

2.3 Characteristics of Geometric Encoding

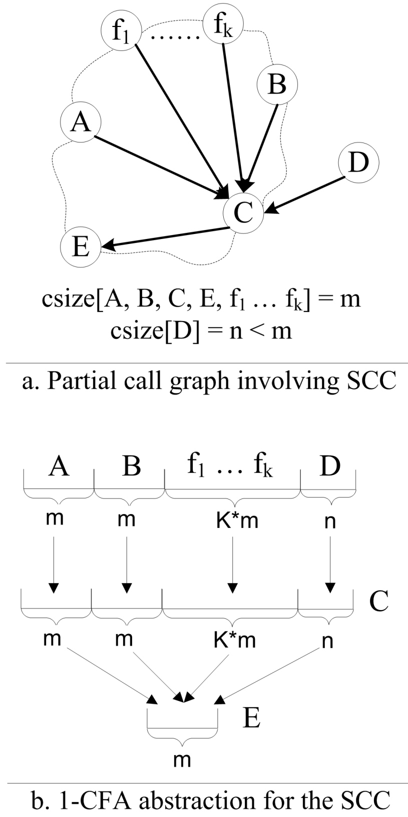
The geometric encoding overall offers higher compression capability and precision fidelity than the state-of-the-art non-BDD based points-to analysis EPA [20]. Without the globals, our encoding is as compact and precise as EPA. This property can be demonstrated through an example of two functions, X 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, ξ_1, o, ξ_2), where the symbols ξ_1 and ξ_2 are the call paths to X and Y descending from Z. In our representation, (p, ξ_1, o, ξ_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(a), since some versions of o^{35} are pointed to by $gList$, the EPA algorithm directly makes o^{35} insensitive, resulting in r points-to both o^{14} and o^{35}. This is because $tail$ and $list1$ are alias under the EPA approach, for which our encoding correctly concludes that they point to different versions of o^{35}. Therefore, in the presence of the globals, our encoding approach is more compact and precise than EPA.

Another noteworthy characteristic of 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 $O(n_1 n_2)$, where n_1 and n_2 are the number of nodes of the input BDDs given to relprod, ranging from $O(1)$ to $O(U)$ $(O(U) = O(2^{\log U}))$.

2.4 The Heap Insensitive Encoding

The use of rectangles complicates the information representation of the geometric encoding. A simpler design is treating the *many-to-many* relation as the *many-to-all* rela-



a. Partial call graph involving SCC

b. 1-CFA abstraction for the SCC

Figure 6: The *1*-CFA modeling of an SCC. A, $\dots$, E, f_1 $\dots f_k$ are functions, and the segments under which denote the context bars.

tion. The term *all* means all the contexts. For example, if we have $(p, o, 1, 3, 5, 7)$ and the context bar of o is $[1, 11)$, we can soundly rewrite this points-to relation as $(p, o, 1, 1, 5, 10)$. The benefit is that, since all the contexts of the interval $[1, 11)$ for o are used, the term "10" can be omitted. And instead, we use digit 0 in the new expression $(p, o, 1, 0, 5)$ as a wildcard for all the contexts of o. This simplification can be understood as making the object *context insensitive*. We refer to this encoding as the HeapIns encoding.

The name HeapIns does not suggest the *1-to-1* and the *many-to-all* relations form a magma. In fact, reasoning with the HeapIns encoding also produces the *all-to-many* and the *all-to-all* relations, in the case of complex constraints instantiation and the assignment from a local to a global followed to another local. However, the inference rules given in Section 2.2.2 are also applicable, because *all contexts* is only a special case of *many* contexts.

2.5 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 SCCs to single nodes and computing the context insensitive results inside of the SCCs [18, 5, 20]. This is problematic 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.

Algorithm 1 produces a *0*-CFA model because we restrict all the members of the same SCC to have the same context sizes, which precludes us from context sensitively treating the call edges to the SCC members. We provide a remedy, the *blocking scheme*, allows the SCC members to have different context sizes. Consider the partial call graph in Figure 6(a), we obtain its *1*-CFA model in Figure 6(b) as follows. We still run Algorithm 1 first and suppose the functions A, B, E and k others in the SCC all have m contexts, and the function D outside the SCC has n contexts ($n < m$). Second, for each function in the SCC, we re-calculate its context size. Taking function C as an example, it has $k+3$ incoming calls, therefore, we let $csize[C] = (k+2) \times m + n$ and divide its corresponding context bar into $k+3$ blocks, where the first $k+2$ blocks have m contexts and the last one has n contexts. Third, we map the call $A \to C$ to the first block, $B \to C$ to the second block, and so on. The final step is mapping C to E, but it is not 1-to-1 any more since $csize[C] > csize[E]$ in our example. Our solution is, we pick that unique block of E and map all the blocks of C to it. This time, we build $k+3$ mappings from C to E for the $k+3$ blocks of C, in contrast to only one mapping for the call $C \to E$ built in Algorithm 1. This treatment resembles to the *1*-CFA context abstraction, because the disambiguated points-to information passed to C from its callers are merged in E. This treatment also answers the question of how to map the call $A \to C$ to the first block of C, which is to map all the blocks of A, if any, to the first block of C.

Algorithm 2 *Points-to_Analysis*()

0. *Constraints_distillation()*;
1. *Build_initial_encoding()*;
2. Worklist $\leftarrow$ pointers have points-to tuples
3. **while** Worklist $\neq \phi$
4.   pick a pointer p from Worklist;
5.   *Geometric_merging()*
6.   **for each** newly added points-to relation $E_{p/o}$
7.   **for each** complex constraint $E_{p/q.f}$
8. $E_{p/o.f}$ = *Instantiate*($E_{p/o}$, $E_{p/q.f}$)
9.   add the edge $E_{p/o.f}$ if it is uncovered
10.   **for each** pointer assignment $E_{p/q}$
11.   **for each** points-to relation $E_{p/o}$
12.   *Propagate*($E_{p/q}$, $E_{p/o}$)
13.   put q into Worklist if $E_{q/o}$ is uncovered
14. **end while**

3. THE POINTS-TO ALGORITHM

Points-to Algorithm. The skeleton of our points-to algorithm is given in Algorithm 2, which only replaces the inference rules of the Anderson's analysis with our new rules given in Section 2.2.2. In summary, we first pick a pointer p from the worklist (Line 4), then we merge the geometric extensions of the points-to and pointer assignments related to p (discussed below). After which, we exploit the newly found points-to facts of p to instantiate the complex constraints (Lines 6-9). Finally, we use the assignment inference rules (Figure 4) to propagate the points-to facts (Lines 10-13).

Our real implementation of Algorithm 2 employs the common acceleration techniques for Anderson's analysis including the difference propagation and the prioritized worklist

[12]. In addition, we use the approaches *constraints distillation* (Line 0) and *geometric merging* (Line 5) to further improve the performance, operated as follows.

Constraints Distillation. The prevalent use of libraries in Java program inhibits the scaling of the points-to analysis. This observation is similar to Rountev's [13]: Most of the library code does not affect the points-to information of the pointers in user's code. It is also similar to the demand driven spirit [16] that not all code is needed for computing the points-to information of a pointer. 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 Rountev's [13] because we know the user's program prior to the analysis. We first identify the pointers of which the 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. We first mark all the pointers appeared in the user classes, then identify all the essential pointers by propagating marks on **the converged assignment graph after Anderson's analysis**. Next, we distill the irrelevant constraints: $q = p$ is irrelevant iff q is inessential, and $q.f = p$ is irrelevant iff all the instance fields $o.f$ instantiated by $p.f$ are inessential.

Geometric Merging. We currently employ the linked list to manage the geometric extensions. Therefore, inserting a new figure into the manager is fast, but the containment testing queries (Line 9, 13) cannot be quickly answered due to the large space of contexts and the sequential searching on the linked list. To improve the performance, we sacrifice the precision manually via the *fractional parameters*, δ_1 and δ_2, to limit the number of geometric extensions every interpreter tuple (V_1, V_2) owns [6]. In Algorithm 2, **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 δ_1 (or δ_2) geometric extensions, and at least one of which is *newly* added to (p, o) (or (p, q)) in the vocabulary of difference propagation [12]. The merged geometric extension is the bounding rectangle of all the shapes described by the extensions before merging, as shown in Figure 7.

Correctness. We prove the termination of Algorithm 2 by contradiction. In the case of infinite geometric extensions are generated, the geometric merging technique finally produces one rectangle covering the whole contexts plane for every interpreter tuple. If this happens, the containment testings in Line 9 and 13 always fail because we already compute all the points-to/assigns-to information. Therefore, worklist will finally be empty and the loop will exit, which contradicts with our assumption. The conservativeness of our algorithm is proved in the following theorem:

THEOREM 3.1. *Algorithm 2 is a safe points-to analysis.*

PROOF. Algorithm 2 consists of four isolated components. The first is the inference rules, the *soundness* of which is guaranteed by Lemma 2.1. The second is our context numbering and mapping model, especially the *1*-CFA model for the SCC. It is *sound* because the essence of the context sensitivity is only a matter of building the mapping between the instances of two functions at every callsite, no matter which instances are chosen and how many instances we abstractly created for each function. The third is the Anderson's analysis framework (using directional assignments, analyzing only four types of constraints, *etc.*), the *safety* of which is commonly accepted. The forth is the geometric merging strategy, its *conservativeness* is guaranteed by two factors: 1). the bounding rectangle always contains all the points-to/assigns-to information before merging; 2). any points-to/assigns-to information used in Lines 9 and 13 to filter the redundant geometric extensions has already been propagated as highlighted in our *geometric merging* discussion. Therefore, we never erroneously intercept any non-propagated information. □

[6]If we have an encoded points-to fact $(p, o, 1, 1, 2)$, we say (p, o) owns the extension $(1, 1, 2)$.

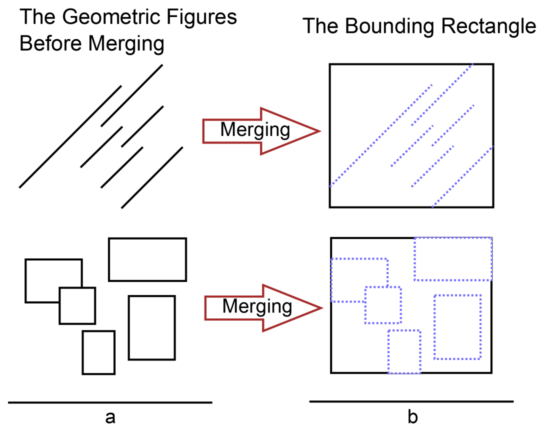


Figure 7: Geometric Merging Illustration. Picture (a) shows the input figures, and picture (b) outlines the corresponding bounding rectangles.

4. EVALUATION

The goal of the experiment is to examine the performance and precision characteristics of our algorithm. We implement our points-to algorithm with the geometric encoding (Geom) and the simplified heap insensitive encoding (HeapIns) in the Soot framework[7], bootstrapped by SPARK [8]. For the purpose to evaluate the impact of the different treatments to SCCs, we implement Geom-0 and Geom-1, standing for the *0*-CFA and *1*-CFA models for SCCs (Section 2.5) with the Geom encoding. In addition, we choose the 1-object-sensitive algorithm [10] implemented in Paddle [6] as the representative of context sensitive analysis[8], and collect the performance data of both their worklist (1-obj-W) and JavaBDD (1-obj-B) based implementations. Some of the 1-obj-B data are missing because the JavaBDD crashed for unknown reasons. Besides the performance comparison, we also assess the precision of our algorithm by the virtual call resolution and the alias analysis, both are fundamental to the high level program analyses.

Experimental Setting. We choose Soot version 2.4.0 as our front-end, and use Sun JDK 1.3.1_20 for Soot as the program analysis base library, while Soot itself is powered by JRokit 28.1 running under the *server* mode and parameterized for minimum garbage collection latency. The exper-

[7]`http://www.sable.mcgill.ca/soot/`

[8]Both the authors of [7] and we consider it provides the best tradeoffs between precision and analysis efficiency.

Table 1: Summary of the benchmarks.

Program	#Contexts	#Methods	Max SCC
jetty	1.1×10^7	2464	853
jlex	2.6×10^7	2534	875
jasmin	1.5×10^7	2695	854
polyglot	1.1×10^7	2453	857
javacup	3.3×10^7	2757	904
jflex	3.9×10^{11}	4081	951
soot	1.5×10^{11}	4697	965
sablecc	1.0×10^{11}	9070	1572
antlr	2.1×10^{11}	3141	910
bloat	4.5×10^{10}	5696	1847
ps	1.6×10^{10}	5660	1419
pmd	$> 9.2 \times 10^{18}$	3556	887
jython	3.1×10^{17}	4231	1408
jedit	8.3×10^8	10266	4965
megamek	8.1×10^{12}	14330	1635

iment is conducted on a 64-bit machine with a Intel Xeon 3.0G processor running Linux kernel 2.6.22. Since minimizing the time usage is our first priority, the minimum/maximum heap size are both set to 15GB in order to reduce the JVM memory acquisition time. Our benchmark includes the programs from the Ashes suite, the Dacapo suite (beta 20050224), and other commonly cited large Java applications. All the benchmark programs are characterized in Table 1. Note that, in terms of the number of analyzed methods (column #Methods), our benchmark size is comparable to the previous work [18, 15] with JDK 1.4.

Implementation. We choose 100 and 50 for the fractional parameters δ_1 and δ_2 (Section 3), which exhibit a good trade-off for performance and precision. To reduce the precision loss, the geometric extensions are never merged for the pointers that have complex constraints, and every allocation site of StringBuffer is treated individually by setting the Soot option merge-stringbuffer to be false. To avoid the use of big integers, the contexts numbered beyond 2^{63} are merged as a single context, similar to Whaley's approach [18]. The user's classes required by the constraints distillation technique are judged by the package name. To be conservative, we only treat the packages in the name spaces of **java**, **javax**, **sun** and **com.sun** as the library code. This assumption is proved by our manual examination.

4.1 Performance

The time and memory usage for all the evaluated algorithms are collected in Table 3. The 1-object-sensitive analysis is running with on-the-fly call graph construction [6], and the running time for HeapIns and Geom-1 exclude the time of SPARK. HeapIns and Geom-1 take only several seconds for the medium and small benchmarks, and 7 minutes for the largest one. In terms of the absolute running time, our algorithms are more practical than the 1-object-sensitive implementation in Paddle. Statistically, HeapIns and Geom-1 are on average 23.9x (min: 1.8x, max: 165.8x) and 11.6x (min: 1.1x, max: 67.8x) faster than 1-obj-W, and the improvements sharply go up to 111.0x (min: 20.5x, max: 174.7x) and 68.3x (min: 13.0x, max: 112.1x), compared to 1-obj-B. To best of our knowledge, the magnitudes of these speedups are rare in the points-to analysis literature.

The memory consumption is also an important factor for measuring the scalability of the points-to algorithms. On average, our algorithms HeapIns and Geom-1 require 2.7x and 2.0x less memory than 1-obj-B and 9.6x and 6.7x less memory than 1-obj-W. Notably, we have 9 out of 15 benchmarks manifest quite low memory usage, which even outperform the BDD based implementation for roughly 3 times. This improvement is significant considering that the most important feature of BDD is memory efficiency.

Table 2: The preprocess time of CDT and the performance of Geom-1 without CDT. The record 5.6 (1.08x) means, jetty needs 1.08x more time to analyze without CDT.

Program	Preprocess Time(s)	Geom-1 Without CDT	
		Time(s)	Mem(MB)
jetty	0.10	5.6 (1.08x)	157 (1.23x)
jlex	0.11	7.1 (1.08x)	172 (0.86x)
jasmin	0.11	6.0 (1.09x)	197 (1.28x)
polyglot	0.10	5.6 (1.06x)	155 (1.34x)
javacup	0.12	6.2 (1.09x)	284 (1.00x)
jflex	0.15	11.7 (1.15x)	527 (1.20x)
soot	0.22	20.6 (1.10x)	614 (1.18x)
sablecc	0.37	98.8 (1.41x)	2259 (1.38x)
antlr	0.13	12.5 (1.09x)	208 (1.28x)
bloat	0.27	134.8 (1.06x)	3415 (1.07x)
ps	0.25	103.1 (1.33x)	1880 (1.34x)
pmd	0.14	64.8 (1.39x)	2486 (1.36x)
jython	0.24	24.1 (1.04x)	625 (1.10x)
jedit	0.51	145.4 (1.39x)	4631 (1.33x)
megamek	0.84	1028.8 (2.55x)	12432 (1.16x)

The efficiency of our algorithms is partly contributed by the constraints distillation technique (CDT). From the column #Constraints of Table 3, we find that CDT effectively deletes 24.4% of the constraints extracted from SPARK. To quantify the impact of those removed constraints, we rerun Geom-1 without CDT and record the time and memory usage in Table 2. From the statistics, CDT reduces 17% on both time and memory on average at the cost of less than 1 second preprocessing time. The time reduction for the largest program megamek is higher than normal, because the memory usage almost achieves the upper bound so that the garbage collector is more frequently invoked.

We believe CDT can be more useful in the case that only a small fraction of pointers need the refined points-to information (*e.g.* the pointers point to a certain objects in Anderson's analysis), in contrast to the whole program in our experiment. For this reason, Table 2 signifies that the main power of our algorithm comes from our new encoding approach combined with the *1*-CFA model for SCC, rather than from CDT. Without the encoding, it is hard for us to outperform the traditional points-to implementation such as Paddle. Moreover, since our encoding efficiently represents the heap cloning, it also indirectly confirms Lhoták and Hendren's conjecture: *Efficiently implementing a 1H-object-sensitive analysis without* BDD *will require new improvements in the data structures and algorithms* [7].

4.2 Precision

Virtual Call Resolution. One application of pointer analysis is to build the *context insensitive call graph* (CICG). A virtual call is *solved* iff we have only one callee for that callsite, where the callees for a callsite are decided by the context insensitive points-to result. Correspondingly, in the *context sensitive call graph* (CSCG), the potential callee for a callsite is decided by the context sensitive points-to result of the base pointer. Since our algorithm is a k-CFA analysis that has a huge number of contexts, genuinely building the

Table 3: Summary of the time and memory usage for all evaluated algorithms.

Program	#Constraints	Time (s)					Memory (MB)				
		SPARK	1-obj-W	1-obj-B	HeapIns	Geom-1	SPARK	1-obj-W	1-obj-B	HeapIns	Geom-1
jetty	23447 (×1.44)	14.3	37.3	520.2	3.1	5.2	293	1486	460	99	115
jlex	26742 (×1.39)	10.8	44.8	550.5	3.8	6.6	299	1641	491	102	129
jasmin	27838 (×1.39)	11.8	43.3	584.5	4.6	5.5	343	1732	509	118	152
polyglot	23495 (×1.44)	15.5	37.9	524.2	3.0	5.3	298	1561	464	96	114
javacup	30279 (×1.35)	11.1	45.9	582.9	4.2	5.7	319	1792	500	220	292
jflex	41827 (×1.4)	19.5	95.3	1143.4	7.1	10.2	418	3928	738	241	444
soot	75209 (×1.2)	17.6	81.9	1226.3	12.9	18.7	410	2430	745	463	631
sablecc	117298(×1.4)	36.8	119.9	1526.7	42.1	70.1	714	3588	845	1027	1561
antlr	35626 (×1.3)	12.6	54.4	720.0	4.4	7.4	335	1990	559	135	162
bloat	95863 (×1.15)	20.8	251.0	2276.1	46.0	126.7	481	5535	858	1450	2989
ps	82477 (×1.35)	25.0	86.4	1003.7	49.0	77.5	517	3215	676	933	1462
pmd	36120 (×1.3)	14.1	65.1	731.0	16.1	45.9	352	2119	579	1193	1886
jython	52873 (×1.2)	17.5	150.9	1236.4	10.4	23.1	407	4139	710	242	631
jedit	119464 (×1.3)	43.0	7078.1	-	42.7	104.4	919	11487	-	1881	3617
megamek	207122 (×1.3)	77.0	14128.7	-	190.0	403.0	1799	9396	-	5807	10223

CSCG is not practical. Instead, we design a *random testing* approach to evaluate the quality of the CSCG. The idea is, for each unsolved callsite in the CICG, we test if we have a better result considering the callsite base pointer context sensitively. In our experiment, we randomly choose 1000 contexts of the base pointer (or choose all the contexts if it has less than 1001 contexts, and normalize the result to 1000 contexts base), and count the number of contexts under which the callsite is solved. The sampling size 1000 faithfully approximates the CSCG, because our conclusion remains for a larger sample size (*e.g.* 5000).

The result of virtual call resolution is given in Table 4. Because different algorithms compute slightly different sets of reachable non-library methods, we take the set obtained by 1-obj-W as the baseline. In the CICG construction, 1-obj-W is relatively better than our algorithms. The gap between Geom-1 and 1-obj-W in the sablecc and bloat benchmarks are notable, because they frequently employ the visitor design pattern and the large number of visitor instances are passed to a few number of pointers from different places[9]. Therefore, Geom-1 often merges the vistor instances before passing to the followers so that the followed pointers get more than expected objects and result in fewer number of callsites resolved. Enlarging the fractional parameters is a way to handle this case. For example, if we let δ_1 and δ_2 equal to 200 and 150 respectively, we can resolve 56 additional virtual callsites than SPARK in the sablecc benchmark, but meanwhile, the computing time is doubled.

Nevertheless, our algorithms show their strong potential in precisely constructing the CSCG. From Table 4, our algorithms outperform 1-obj-W on all benchmarks except pmd, and the gap in some cases, *e.g.* jflex, jython and ps, are quite significant. This is desired because the higher k in contexts abstraction, the finer the points-to information under each context. This is akin to mapping all the points in a 3D space to a 2D space, then every grid unit in the 2D space must have the same number or more points compared to its corresponding grid units in the original 3D space. Back to our problem domain, the points-to information scattered in 3D space is the solution of Geom-1, while the 2D space counterpart describes the 1-obj-W result.

HeapIns exhibits better precision in the CSCG construction and in the cases sablecc and bloat of the CICG, compared to Geom-1. This is because, the termed *all* relations (*e.g. many-to-all*) make the number of geometric extensions owned by each interpreter tuple grow slower, resulting in the less frequent invocation of geometric merging and finally leading to less uncertainty in precision. But the HeapIns encoding is a double-edged sword. In applications that demand the heap sensitivity, HeapIns would perform worse than Geom-1 in, for example, the alias analysis.

Alias Analysis. In our experiment, we leverage the widely accepted *all-pairs-alias* methodology proposed in [2, 5] to assess the quality of alias analysis. Precisely, we scan a user function, and collect all the pointers p and instance fields $o.f$ accessed in the function, if the type of that pointer or field is not a sub-class of java Exception. Then, we exhaustively enumerate two pointers p, q and intersect their points-to sets, with different instances of the same object disambiguated and constants ignored if possible.

The alias analysis result is presented in Figure 8. We take the number of alias pairs in SPARK as the baseline, and the alias analysis quality for each context sensitive algorithm is characterized as the percentage of alias pairs produced by SPARK. Overall, the three algorithms 1-obj, HeapIns and Geom-0 perform closely well: They respectively reduce 15.5%, 16.8% and 16.0% alias pairs made by SPARK. However, the precision of Geom-1 is dramatically better, it negates 21.2% alias pairs of SPARK. This is the achievement of the heap cloning and *1*-CFA model for SCCs. Moreover, since our algorithms require much less computing time and memory, this improvement is noteworthy and significant.

4.3 Impact Study of Recursive Calls

We have noticed the power of *1*-CFA model for SCCs in the alias analysis. In this section, we try to quantify the influences of SCCs, *i.e.* how much precision we gain and meanwhile how much extra time we need if any. To study the precision, we measure the *average context insensitive points-to pairs per pointer* (APP) or *average var-points-to* [15] in user's code. APP is a good estimation of how many pointers are affected by the SCCs because it counts all the pointers once and only once. However, APP is not a good predictor of the precision gain for the client applications evaluated previously. For example, Geom-0 performs closely well to Geom-1 in the call graph construction, because most of the user functions involved in the SCCs are the overloaded versions of *toString* and *printStackTrace etc.*, which are ir-

[9]It is the feature of the visitor pattern: How to modify the data is decided by the input visitors. Therefore, the callsites base pointers are the limited several.

Table 4: Virtual Call Resolution. CICG reports the number of virtual callsites resolved in context insensitive call graph, and the rests show the relative numbers to the SPARK column. CSCG reports the percentage of additional virtual callsites resolved in context sensitive call graph.

Benchmark	Total	CICG					CSCG (%)			
		SPARK	1-obj-W	HeapIns	Geom-0	Geom-1	1-obj-W	HeapIns	Geom-0	Geom-1
jetty	281	267	+0	+0	+0	+0	0	0	0	0
jlex	772	771	+0	+0	+0	+0	0	0	0	0
jasmin	947	933	+0	+0	+0	+0	13.5	14.2	4.2	14.3
polyglot	233	232	+0	+0	+0	+0	0	0	0	0
javacup	2023	1978	+0	+0	+0	+0	1.7	8.9	7.8	8.9
jflex	2580	2566	+0	+0	+0	+0	0.0	56.8	53.6	52.5
soot	5723	5384	+0	+1	+1	+1	16.7	39.0	44.7	39.0
sablecc	3953	3648	+55	+53	+16	+16	26.4	44.5	36.0	36.1
antlr	4419	4032	+1	+1	+1	+1	1.1	3.0	2.9	3.0
bloat	12479	11868	+27	+21	+8	+11	17.3	34.0	46.1	33.6
ps	2338	2032	+3	+1	+3	+3	0.3	9.9	9.8	9.9
pmd	1985	1975	+0	+0	+0	+0	26.7	13.0	15.2	14.1
jython	5963	5652	+1	+1	+1	+1	19.3	55.4	30.2	54.9
jedit	11136	10774	+3	+1	+1	+1	9.8	37.1	49.7	36.6
megamek	31632	30605	+90	+1	+90	+90	24.5	32.8	23.2	26.1

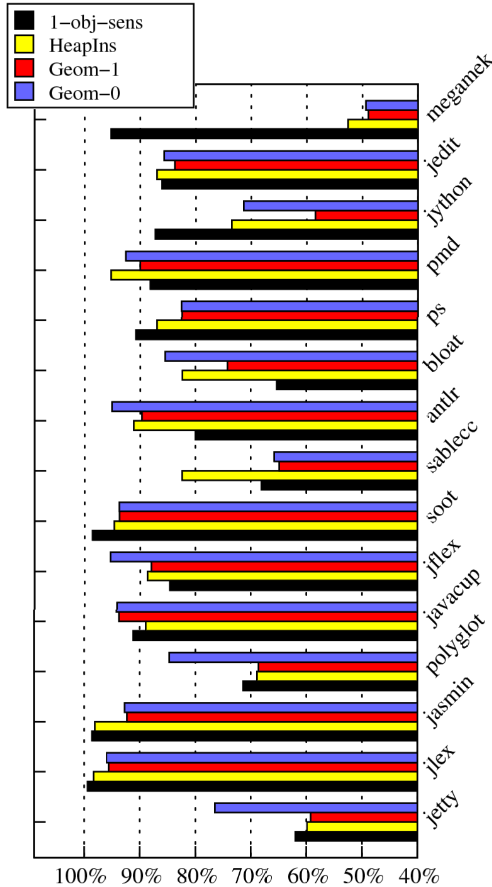


Figure 8: Alias analysis. Each bar stands for the percentage of alias pairs of that computed by SPARK.

relevant to the virtual callsites.

Table 5 collects our experimental data. We observe that, Geom-1 improves the APP metric of Geom-0 on all benchmarks. On average, the reduction is 37%; but it is dramatic in the cases jetty, polyglot and antlr, all of which exhibit more than 77% reduction. The reduction is roughly estimable by the Max SCC metric in Table 1. This is because that, the more user functions are affected by the SCCs, the higher precision we gain. The time usage is nearly constantly reduced and, on average, Geom-0 takes 2.9x more computing time. This is desirable because less points-to relations lead to less propagation rounds of matched facts [15]. But, meanwhile, as pinpointed by Smaragdakis *et.al* [15], Geom-1 will be less scalable when imprecision will occur anyway (*e.g.*, through spurious assignments, flow insensitivity, geometric merging and blocking scheme) in some cases (*e.g.*, pmd).

Table 5: Comparison of the *0*-CFA and *1*-CFA model for recursive calls. The data in bold are those larger than the corresponding Geom-1 results.

Program	Time(s)	Mem(MB)	Avg. Points-to Pairs	
			Geom-0	Geom-1
jetty	**37.4**	**1500**	31.5	7.2
jlex	**38.5**	**1511**	18.1	9.9
jasmin	**32.7**	**1450**	33.8	25.3
polyglot	**33.6**	**1410**	35.7	7.4
javacup	**14**	278	99.7	79.3
jflex	**29.3**	**582**	107.1	66.1
soot	**37**	**665**	62.8	47.1
sablecc	**131.8**	**1908**	46.1	27.6
antlr	**13.3**	**260**	38.3	6.8
bloat	**188.5**	2912	139.4	126.2
ps	74.9	1459	149.4	141.0
pmd	8.0	464	27.6	14.4
jython	**41.8**	**907**	72.0	44.1
jedit	**131.5**	2645	68.7	54.0
megamek	**589.3**	8565	99.0	95.0

Predicting the memory increment of Geom-0 over Geom-1 is harder because it is influenced by two opposing forces. One is the increase of the spurious points-to facts, and the other is the decrease of the geometric figures owned by each interpreter tuple due to geometric merging. Both forces cause the computing time to increase, but they have opposite effects for the memory usage and we cannot tell which force has the upper hand for an arbitrary program. For example, the first force prevails in the first four benchmarks, but the second force wins in the last two. In some cases where the *1*-CFA model requires more memory, it only requires 22% more. In the cases where the first force prevails, the memory reduction is 86%, which is more significant.

5. RELATED WORK

Points-to analysis is a well studied subject of a large body

of work. We choose to discuss the most related and recent ones in the category of designing or engineering an extremely fast yet still precise points-to analysis. One of them, **bddbddb**, is a highly flexible engine designed by Whaley *et al.*, aimed for prototyping a range of program analysis algorithms [18]. The points-to algorithm 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 [7], the precision of bddbddb is only comparable to the 1-callsite analysis. We cannot directly compare our algorithm with bddbddb, because their compiler models the program in a different way from Soot [1]. But as shown by Lhoták [7], the Paddle's version of Whaley's algorithm is slower than the 1-object-sensitive analysis, which is an evidence to show our performance superiority over bddbddb.

Doop [1] implements a range of points-to analyses in declarative language with rich features (*e.g.* declarative on-the-fly call graph construction) and high performance. Similar to our work, Doop's high performance is also obtained from non-BDD based constraints evaluation. However, it's points-to and pointer assignments information is explicitly stored without compaction. Therefore, Doop is limited to support k-CFA analysis with larger k. It is interesting to compare with Doop in future if we could access to its commercial Datalog engine.

The EPA algorithm [20] 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. In fact, the EPA algorithm has a sophisticated implementation that we did not manage to successfully port it in our experimental setting. Therefore, the comparison with EPA is provisionally absent in this paper. But our algorithm does not compute the escaped objects, instantiate and merge the symbolic objects, and remove the context information for the objects pointed to by global variables, thus, we expect our algorithm to be much faster. Of course, a fair comparison is needed to prove our hypothesis.

6. 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 68x faster than the BDD based 1-object-sensitive analysis, meanwhile their precision is similar in the call graph construction and we are better in the alias analysis. Our future work will incorporate the on-the-fly call graph construction, and experiment our encoding for the object-sensitivity and the callsite and object combined abstractions [15], to see if the geometric encoding is a powerful backbone for these practical models. 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 the practical full context sensitive points-to analysis.

7. ACKNOWLEDGEMENTS

We sincerely thank the anonymous ISSTA reviewers especially our shepherd for their insightful feedback. This research is supported by RGC GRF grants 622208 and 622909.

8. REFERENCES

[1] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA '09*. ACM.

[2] M. Das, B. Liblit, M. Fähndrich, and J. Rehof. Estimating the impact of scalable pointer analysis on optimization. In *SAS '01*. Springer.

[3] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI '94*. ACM.

[4] B. Hardekopf and C. Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI '07*. ACM.

[5] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. *PLDI '07*.

[6] O. Lhoták. *Program analysis using binary decision diagrams*. PhD thesis, Montreal, Canada.

[7] O. Lhoták and L. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. *ACM TOSEM*, 2008.

[8] O. Lhoták and H. Laurie. Scaling java points-to analysis using spark. volume 2622. Springer.

[9] D. Liang and M. J. Harrold. Efficient computation of parameterized pointer information for interprocedural analyses. In *SAS '01*.

[10] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM TOSEM*.

[11] E. M. Nystrom, H. seok Kim, and W. mei W. Hwu. Importance of heap specialization in pointer analysis. In *PASTE '04*. ACM.

[12] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Online cycle detection and difference propagation for pointer analysis. In *IEEE SCAM '03*.

[13] A. Rountev and B. G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *CC '01*.

[14] O. G. Shivers. *Control-flow analysis of higher-order languages of taming lambda*. PhD thesis, Carnegie Mellon University.

[15] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: Understanding object-sensitivity. In *POPL '11*. ACM.

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

[17] J. Whaley. *Context-Sensitive Pointer Analysis using Binary Decision Diagrams*. PhD thesis, Stanford University, Mar. 2007.

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

[19] R. P. Wilson and M. S. Lam. Efficient context sensitive pointer analysis for c programs. In *PLDI '95*.

[20] G. Xu and A. Rountev. Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis. In *ISSTA '08*. ACM.