EPA: A Precise & Scalable Object-Sensitive Points-to Analysis for Large Programs

Behnaz Hassanshahi[1,2], Raghavendra K.R.[2]
Padmanabhan Krishnan[2], Bernhard Scholz[2,3], and Yi Lu[2]

[1] School of Computing,
National University of Singapore,
Singapore.
b.hassanshahi@u.nus.edu
[2] Oracle Labs,
Brisbane.
{raghavendra.kr,paddy.krishnan,yi.x.lu}@oracle.com
[3] School of Information Technologies,
University of Sydney,
Sydney.
bernhard.scholz@sydney.edu.au

Abstract. Points-to analysis is a fundamental static program analysis technique for tools including compilers and bug-checkers. There are several kinds of points-to analyses that trade-off precision with runtime. For object-oriented languages including Java, "context-sensitivity" is key to obtain sufficient precision. A context may be parameterizable, and may consider calls, objects, types for its construction. Although points-to analysis research has received a lot of attention in the past, scaling object-sensitive points-to analysis for large Java code bases still remains an open research challenge.

In this paper, we develop an *Eclectic Points-To Analysis* (EPA) framework that computes an efficient, selective, object-sensitive points-to analysis that is client independent. This framework parameterizes context sensitivities for different allocation sites in the program. The level of required sensitivity is determined by a pre-analysis.

We have implemented our approach using Soufflé (a Datalog compiler) and an extension of the DOOP framework. Our experiments on large programs including OpenJDK[4] and Jython show that our technique is efficient and highly precise. For the OpenJDK, an instance of the EPA-based analysis reduces 27% of runtime for a slight loss of precision, while for Jython, the same analysis reduces 82% of runtime for almost no loss of precision.

[1] [3] Work done while visiting Oracle Labs, Brisbane.

[4] Java and JDK are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

1 Introduction

There are many applications for the points-to analysis in tools and compilers including taint and escape analyses, compiler optimizations and specific client analyses, such as detecting potential security vulnerabilities. Hence, points-to analysis is a fundamental building block that is essential for programming language tools.

Points-to analysis is a static program analysis technique that builds a heap abstraction of the input program without executing it. The key insight in points-to analyses frameworks is that objects are abstracted by the their object-creation sites [And94], reducing a potentially infinite set of objects to a finite set of object-creation sites of a given input program. This abstraction makes the points-to analysis computable.

A heap abstraction is an over-approximation of all observable heap states of the input program. Most points-to approaches for object-oriented languages, use two functions to form a heap abstraction. The first function, maps variables to sets of object-creation sites and the second function maps object-creation sites with a given field in an object to other sets of object-creation sites [SGSB05].

The heap abstraction is also used to tighten the potential virtual methods that might be called at a invocation site. This is achieved by wiring methods that are invoked for the types found in the points-to sets of the receiver variables of invocation site. A points-to analysis that tightens the set of virtual methods using the points-to analysis refectively, is also *call-graph construction on the fly* [SBL11]. Call-graph construction on-the-fly improves the precision of points-to analyses (rather than wiring naïvely all potential methods without considering the points-to sets of the receiver variables).

Adding context sensitivity to points-to analyses has been another method to improve the precision. Context-sensitivity is introduced by versioning variables and object-creation sites. Each version is attached to a context. There are many kinds of context sensitivities including call-site-based, object-based and type-based. Object-based context sensitivity (object sensitivity) is shown to be the most effective context sensitivity in terms of both precision and computational costs for object-oriented programs when compared to call-site-based and type-based context-sensitive points-to analysis [MRR05,SBL11].

Object-sensitive points-to analysis uses objects as contexts for both invocations and object allocations. For invocations, it qualifies each method invocation with a finite ordered sequence of objects starting from the receiver object of the method, say o_1, then the object that created o_1, say o_2, then the object that created o_2 and so on. Using objects as contexts for method invocations is the method cloning idea [WL04] for object-oriented programs based on the sequence of objects. Typically, object-sensitive points-to analysis also uses a form of heap cloning [LLA07] in the context of object creation. Here, the heap cloning means that the allocation sites are also qualified by a sequence of objects starting from the creator object, say o_1, creator of o_1, say o_2, and so on.

However, applying a sufficiently precise object-sensitive points-to analysis uniformly to large programs is costly in with respect to the runtime and mem-

ory consumption. We observe that if the chosen heap cloning context is not sufficient for a particular object, the resulting points-to relation has many spurious facts for other related objects. This choice has a cascading effect on other objects, resulting in valuable resources being spent on computing irrelevant and unnecessary facts. The option of just increasing the level of object sensitivity for all allocation sites in an attempt to improve precision is not always feasible because it typically incurs unacceptably high computational costs without corresponding benefits.

In this paper, we address the following research problem. Given a program P, identify the allocation sites in P that compound the generation of spurious points-to facts, and determine the object-sensitivity depth required for each such allocation site in P so as to make the points-to analysis scalable and precise.

To resolve these problems, we have developed an *Eclectic Points-To Analysis* (EPA), a framework for scaling object-sensitive points-to analysis to large codebases while maintaining high precision with two stages—pre-analysis and main analysis. The pre-analysis consists of four steps.

1. Extract a portion (which we call *kernel*) from the given input program. The kernel comprises program elements that are bottlenecks when applying the fixed object-sensitive points-to analysis on the input program.
2. Perform fixed object-sensitive points-to analysis on kernel.
3. Identify the allocation sites in the kernel along with their contexts that are not effective in removing spurious points-to relations.
4. Calculate requisite depths of object sensitivity needed for these allocation sites so as to maintain both good precision and computational efficiency.

The main stage performs a *selective* object-sensitive points-to analysis is performed on the input program with the previously determined object-sensitivity depths for specific allocation sites.

The main contributions in the EPA framework are the following:

1. A technique based on context-insensitive points-to analysis (which is inexpensive but very imprecise) to extract the kernel, i.e., a portion of the input program for which fixed object-sensitive points-to analysis is computationally expensive compared to the rest of the input program.
2. Novel metrics based on fixed object-sensitive points-to analysis on the kernel that determine the allocation sites where the contexts are insufficient or where contexts do not add much value.
3. A heuristic to identify the context-depths that will overcome the efficiency bottleneck with/without losing precision of fixed object-sensitive points-to analysis. That is, the metrics in the previous step are used to identify allocation sites where contexts need to be be increased as well as to identify allocation sites where contexts can be reduced without decreasing precision significantly.

The rest of this paper is organised as follows. Section 2 establishes definitions and preliminaries required to understand the technicalities of our approach. Section 3 discusses an example that motivates the problem in detail and outlines

our approach. Section 4 describes our technique in detail. Section 5 describes our implementation and experimental results. Section 6 surveys the related work in the literature and contrasts them with our approach. Section 7 concludes the paper by summarising our contributions and pointing to future directions. future work.

2 Preliminaries

In this section, we summarise the key concepts and definitions that are used in the sequel of the paper. We use points-to analysis of the DOOP [BS09] framework and optimize it to Souffle Datalog engine [SJSW16]. DOOP framework uses the domain of different sets like the set of program variables, object allocation sites, method invocation sites, field names etc. The input program is represented as relations on these domains. From these relations, the points-to analysis computes another set of relations. For a detailed definitions, please refer to [SB15a].

Input Relations. We first describe the relations that capture the structure of the input program. We assume that the domain of classes (or types), methods and variables are defined.

$Alloc(o, m)$ indicates that there is an allocation site labelled o in method m.
$Store(b, f, v)$ indicates that there is an assignment of the form $b.f \leftarrow v$ where b and v are variables and f is a field. We overload this relation for storing elements in an array where f represents the index. However, as we do not distinguish the different elements in the array, the field f will represent a wild-card.
$Load(b, f, v)$ indicates that there is an assignment of the form $v \leftarrow b.f$ where b and v are variables and f is a field. As in the case of *Store* the field f will represent a wild-card in case of arrays.

Computed Relations. Our framework computes the following relations and uses them in different stages of the analysis.

$PointsTo(hc, h, c, v)$ is the result of object sensitive points-to analysis. It states that the variable v under the object context c points to an object h qualified with the heap context hc.
$PointsTo(h, v)$ is the result of context-insensitive points-to analysis. It states that the variable v points to an object h. We overload the same relation name both for the results of context insensitive and object sensitive points-to analysis.
$PointedToByVar(o)$ is the set $\{v \mid PointsTo(o, v)\}$.
$MethodPointsTo(m)$ is the set $\{o \mid PointsTo(o, v)$ for a variable v declared in $m\}$.
$ReachableCtx(m, c)$ states that method m is reachable in context c from the program's main entry/entries.

Object Sensitivity. The above definitions do not specify the exact contexts used in the analysis. To define a general n_1O+n_2H object-sensitive points-to analysis [SKB14], the terms n_1O and n_2H have the following meaning.

n_1O**: n_1-Method cloning.** An instance method m which is a potential target of an invocation is distinguished with respect to an ordered sequence of objects of the form $(o_1, o_2, \ldots, o_{n_1})$ where o_{n_1} is the base object of the invocation of m, o_i is the object creating o_{i+1}. We often refer to this context as the object context.

n_2H**: n_2-Heap cloning.** A heap object o is distinguished with respect to an ordered sequence of objects of the form $(o_1, o_2, \ldots, o_{n_2})$ where o_{n_2} is the object creating o, o_j is the object creating o_{j+1}. We often refer to this context as the heap context.

3 Motivating Example

In this section, we give an example that illustrates how some spurious contexts lead to generation of many more spurious contexts. We then make a case that selective object sensitive points-to analysis is better than fixed object sensitive points-to analysis.

The example program is given in Fig. 1. The points-to analysis results for that program are shown in Table 1. We use the notation $(hc, o) \leftarrow (c, v)$ to indicate that the variable v in the context c points to the o in the heap context hc. The number of elements in each hc and c depends on the chosen depth.

2O+2H	**3O+3H**	**Selective**
([o_3,o_2],o_1)←([o_3,o_2],arr)	([o_4,o_3,o_2],o_1)←([o_4,o_3,o_2],arr)	([o_4,o_3,o_2],o_1)←([o_4,o_3,o_2],arr)
	([o_6,o_3,o_2],o_1)←([o_6,o_3,o_2],arr)	([o_6,o_3,o_2],o_1)←([o_6,o_3,o_2],arr)
([o_4,o_3],o_2)←([o_4,o_3],c)		
([o_6,o_3],o_2)←([o_6,o_3],c)	([o_8,o_4,o_3],o_2)←([o_8, o_4,o_3],c)	([o_4,o_3],o_2)←([o_8, o_4,o_3],c)
	([o_8,o_6,o_3],o_2)←([o_8, o_6,o_3],c)	([o_6,o_3],o_2)←([o_8, o_6,o_3],c)
([-,o_8],o_5)←([o_3,o_2],x)		
([-,o_8],o_7)←([o_3,o_2],x)	([-,-,o_8],o_5)←([o_4,o_3,o_2],x)	([o_8],o_5)←([o_4,o_3,o_2],x)
	([-,-,o_8],o_7)←([o_6,o_3,o_2],x)	([o_8],o_7)←([o_6,o_3,o_2],x)
([-,o_8],o_5)←([o_3,o_2],ret)		
([-,o_8],o_7)←([o_3,o_2],ret)	([-,-,o_8],o_5)←([o_4,o_3,o_2],ret)	([o_8],o_5)←([o_4,o_3,o_2],ret)
	([-,-,o_8],o_7)←([o_6,o_3,o_2],ret)	([o_8],o_7)←([o_6,o_3,o_2],ret)
([-,o_8],o_5)←([o_4,o_3],y)		
([-,o_8],o_5)←([o_6,o_3],y)	([-,-,o_8],o_5)←([-,o_4,o_3],y)	([o_8],o_5)←([-,o_4,o_3],y)
([-,o_8],o_7)←([o_4,o_3],y)	([-,-,o_8],o_7)←([-,o_6,o_3],y)	([o_8],o_5)←([-,o_4,o_3],y)
([-,o_8],o_7)←([o_6,o_3],y)		

Table 1: Points-To results for the example program

```
public class Container{
    o1: arr = new Object[10];
    int index=0;
    public void put(Object x){
        arr[index++] = x;
    }
    public Object get(int i){
        ret  = arr[i];
        return ret;
    }
}
public class C1{
    public void m1(Object a1){
        o2: Container c = new Container();
        c.put(a1);
        y = c.get(1);
    }
}
```

```
public class C2 {
    public void m2(Object a2) {
        o3 : C1 d = new ...;
        d.m1(a2);
    }
}
public class C3 {
    public void m3(){
        o4: C2 a = new C2();
        o5: Object x1 = new ...;
        a.m2(x1);
        o6: C2 b = new C2();
        o7: Object x2 = new ...;
        b.m2(x2);
    }
    public void main() {
        o8: C3 a = new C3();
        c.m3();
    }
}
```

Fig. 1: Example Program

Here, we have an example of arrays used in a Java container class. In this example, the program element that is the source of imprecision is `arr`, which has array type. A precise points-to analysis for data structures in general (and arrays specifically) is known to be challenging. Therefore, a common approach for modeling this language feature is array-index insensitivity [SB15b]. The rules used to compute array points-to do not distinguish the load and stores to different array locations which results in imprecision.

The results in Table 1 demonstrate that the 2O+2H context sensitivity is not enough for distinguishing points-to tuples loaded from the array to variable `y` in method `m3`. Note that there are two spurious points-to tuples shown in the first column and indicated in bold font. The results from 3O+3H remove these spurious tuples because the variable y does not point to either o_5 in the context of $[o_6, o_3]$ or o_7 in the context of $[o_4, o_3]$. Thus, increasing context depth adds value.

The results in Table 1 (third column) further show that the heap contexts qualifying the objects o_2 can be shortened by removing o_8. Similarly, the heap contexts qualifying the objects o_5 and o_7 may also be shortened. These results show that the decreasing context depths also add value in terms of not associating useless contexts, and thus increases efficiency.

Therefore, an algorithm, which can estimate the effectiveness of a chosen context sensitivity (such as 2O+1H), and selectively apply deeper contexts where

it is necessary or likely to be necessary, is essential. Our main contribution is such an algorithm that identifies the different heap context depths for each allocation site.

4 Eclectic Points-To Analysis (EPA)

In this section, we describe our general eclectic object-sensitive points-to analysis (EPA) framework. Fig. 2 describes the key steps and the workflow in our framework. The approach can be separated into two stages: pre-analysis and main analysis.

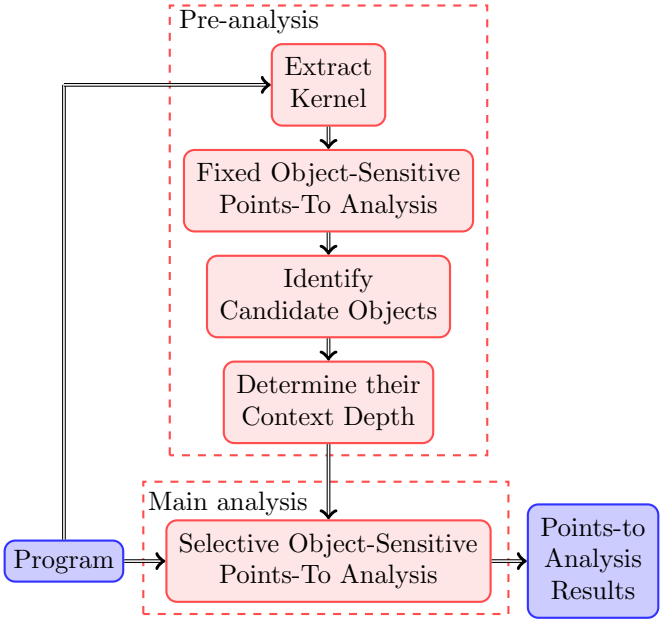

Fig. 2: Workflow in Eclectic Points-To Analysis

The pre-analysis stage identifies the heap allocation sites that are potential bottlenecks when applying a chosen fixed object-sensitive points-to analysis uniformly on the input program. It also determines the requisite depth of object sensitivity needed for these allocation sites. The main analysis uses the information gathered in pre-analysis (i.e., the selective sensitivity for each allocation site) and performs the points-to analysis.

We first describe the four steps in the pre-analysis in details.

4.1 Extracting the Kernel

The first step is to extract the portion of the input program that contains the allocation sites that possibly generate spurious points-to information from a fixed object-sensitive points-to analysis. We refer to this portion of the input program as the kernel.

How do we extract the kernel? It is done using the results of a computationally inexpensive, context-insensitive, Anderson-style points-to analysis on the input program. Given thresholds K_1 and K_2, a class c of the input program is *selected* if:

- there is an object o of class c with $|PointedToByVar(o)| \geq K_1$, or
- there is an object o allocated in a method of class c with $|PointedToByVar(o)| \geq K_1$, or
- there is a method m in class c with $|MethodPointsTo(m)| \geq K_2$.

Then the kernel is constructed from the input program by removing all objects except the selected classes. The intuition here is that for the remaining portion of the program, a context-insensitive or a fixed object-sensitive analysis is sufficiently precise.

Note that if there is even a single object with $|PointedToByVar(o)| \geq K_1$, the entire class identified by o and all its related objects will be retained. Also note that the kernel may not be a valid program but has the program elements that can make context-sensitive points-to analysis less precise.

The thresholds K_1 and K_2 may be chosen manually using the empirical data of multiple experiments that characterize the given set of benchmarks. One heuristic for estimating K_1 and K_2 is based on statistical estimation as depicted in Algorithm 1, which uses the context-insensitive points-to analysis results and a timeout limit. It uses the mean sizes of $PointedToByVar$ and $MethodsPointsTo$ as the initial estimate and runs the fixed context-sensitive points-to analysis. If the computation of this points-to analysis exceeds the resource bound (indicated by the timeout *limit*), the estimates are deemed to be ineffective and are increased iteratively.

4.2 Fixed Object-Sensitive Analysis

The second step (as shown in Fig. 2) is to perform a fixed object-sensitive points-to analysis on the kernel. For this step, we use the standard $mO + nH$ points-to analysis of the DOOP [BS09,SBL11] framework. Note that the fixed object-sensitive points-to analysis on the original program P is expensive but applying it on the kernel is not, as only a subset of the objects in the original program is retained.

4.3 Identifying Candidate Objects

The third step (as shown in Fig. 2) is to identify the candidate allocation sites from the results of the fixed object-sensitive points-to analysis on the kernel

Algorithm 1 Parameter Estimation

Input: A program P, *results*, *limit*
Output: Values K_1 and K_2

```
function EstimateParameters(P, results)
    K1 = mean(0, max(|PointedToByVar|)
    K2 = mean(0, max(|MethodsPointsTo|)
    while Ineffective(K1, K2) do
        K1 = mean(K1, max(|PointedToByVar|)
        K2 = mean(K2, max(|MethodsPointsTo|)
    end while
    return (K1, K2)
end function

function Ineffective(K1, K2)
    kernel = ExtractKernel(P, K1, K2)
    if Timeout(limit, FixedCSPointsTo(kernel) then
        return True
    else
        return False
    end if
end function
```

where the contexts are not effective in removing spurious points-to tuples. Objects in the kernel that potentially have the *compounded smashing effect*, as we call it, are selected. Typically, certain program elements are not handled precisely. For example, the elements of the arrays are *smashed*, i.e., they are not distinguished in a simple points-to analysis. That means, no matter what, we already have some amount of spuriousness. Adding context sensitivity to analyse such program elements may have a cascading effect, leading to many spurious points-to facts being generated. The reason could be either insufficient context sensitivity or context sensitivity is not the way to precisely handle program elements with compounded smashing effect.

4.4 Determinining Context Depths

The fourth and final step in the pre-analysis determines the requisite object sensitivity depth for the candidate allocation sites selected in the previous step. To help describe this process, we define some terms and metrics.

InFlow For a given object o, and field f, $InFlow_f(o)$ gives a measure on the heap contexts related to the heap objects that are stored in the field f of object o, as given by the fixed object-sensitive points-to analysis on the kernel. Recall that for arrays, the field f is ignored. To simplify the presentation we ignore the field f in all cases – but technically the *Load* and *Store* pairs are matched up via the named field f. Thus we use $InFlow(o)$ in our presentation.

$$InFlow(o) = \left\{ (h, hc, oc) \;\middle|\; \begin{array}{l} Store(b, *, v) \text{ for some variables } b, v \\ PointsTo(oc, o, c, b) \text{ and} \\ PointsTo(hc, h, c, v) \text{ for some context } c \end{array} \right\}$$

An example of *InFlow* is shown in Fig. 3. The left side of the figure shows the variable v in some context c pointing to three different objects and the variable b in the same context pointing to two different objects. The right side shows the result of the $Store(b, *, v)$ operation, where the fields or member elements are smashed, indicated by $*$. This smashing causes each of the objects that b was pointing-to to now point to all the objects that v earlier pointed to. That is, the points-to results for both of o_1 and o_2 are updated with $\langle hc_1, h_1\rangle$, $\langle hc_2, h_2\rangle$ and $\langle hc_3, h_3\rangle$. Spuriousness in the points-to of b causes many spurious points-to results after the store operation, which is an example of compounded smashing effect—the larger the set *InFlow* of an object o, the greater the effect of spuriousness. Hence, we investigate objects o whose $|InFlow(o)|$ is greater than a threshold K_4. The value of K_4 depends on the set of benchmarks and can be empirically estimated. We treat K_4 as an input parameter for our framework.

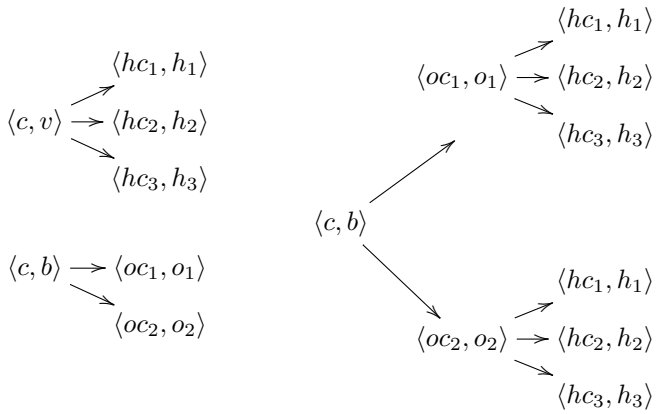

Fig. 3: Example of InFlow

OutFlow For a given variable v, object o, field f and a context oc, $OutFlow_f(v, o, oc)$ gives a measure on the heap contexts related to the heap objects that are loaded from the field f of the object o qualified with context oc for v. But as in the case of *InFlow*, we ignore the field f to simplify the explanation.

$$OutFlow(v, o, oc) = \left\{ (c, h, hc) \;\middle|\; \begin{array}{l} Load(b, *, v) \text{ for some variable } b \\ PointsTo(oc, o, c, b) \text{ and} \\ PointsTo(hc, h, c, v) \end{array} \right\}$$

The intuition behind *OutFlow* is analogous to *InFlow* and captures the cascading or multiplier effect of the *Load* operation. However, note that this set is defined per (loaded) variable and per heap cloning context.

ContextValue For an object o and a context oc, $ContextValue(o, oc)$ estimates the value of the heap context oc generated for o. In other words, it determines the effectiveness of the context generated for an object. More precisely, we define

$$ContextValue(o, oc) = \min_{v} \frac{|InFlow(o)| \cdot |CtxInOutFlow(v, o, oc)|}{|OutFlow(v, o, oc)|}$$

where

$$CtxInOutFlow(v, o, oc) = \{c \mid (c, h, hc) \in OutFlow(v, o, oc) \text{ for some } h, hc\}.$$

Intuitively, for an object o with sufficiently large *InFlow*, if $ContextValue(o, oc) = 1$, it means that the context oc has not distinguished any points-to facts. If $ContextValue(o, oc)$ is greater than a threshold, say K_3, then context oc is valuable as it has distinguished some points-to facts. Similar to K_4, we treat K_3 also as an input parameter for our analysis.

We now design an oracle to determine the necessary depth of heap-cloning, object-sensitive contexts by using the metric *ContextValue*. The depth can range from 0 (representing context-insensitive analysis) to the maximum desired value $k + 1$ (input to our framework). This oracle is described in Algorithm 2.

Algorithm 2 combines the steps of fixed object-sensitive analysis, identifying candidate objects and determining context depths. The thresholds K_3, K_4 and the maximum heap-cloning object sensitivity to be explored k are taken as inputs. Line 2 computes fixed object-sensitive points-to analysis, say, of the order $mO + nH$, on the kernel. Function *ComputeMetrics* in line 3 computes the sets *InFlow*, *OutFlow* and the metric *ContextValue* using the fixed object-sensitive points-to analysis results. Line 4 identifies the candidate objects along with their heap contexts (o, oc) in the kernel, which can have compounded smashing effect. If $|InFlow(o)|$ is smaller than K_4 then we retain the current heap context depth for o in line 17. Similarly we retain the context of the fixed object-sensitive analysis if the $ContextValue(o, oc)$ is greater than K_3, as it is already adding good value. Otherwise, we check if extending the context to depth $k+1$ adds any value using *ContextCorrelation* in line 6. If so, we set the *depth* parameter for those objects to $k + 1$. To generate the context depth of $k + 1$ for such an object, we may need to generate deeper contexts for other objects. This context generation is captured in the sequence of statements in lines 8 to 12. If we determine that it is not useful to extend the context using *ContextCorrelation* then we switch off the context sensitivity for that object in line 14, as we already know that the current context oc is also not adding value. One could choose to not switch off the context sensitivity and have a purely increasing context sensitivity also.

Algorithm 2 Determining the required object-sensitivity depth

Input: kernel, thresholds K_3, K_4, bound on object sensitivity k with $k > n$
Output: *depth*, an associative array

procedure DETERMINECONTEXTDEPTH(kernel, K_3, K_4, k)
    results = *mO+nH-object-sensitive-points-to*(kernel)
    (*InFlow*, *OutFlow*, *ContextValue*) = *ComputeMetrics*(*results*)
    for every (o, oc) in kernel with *compounded smashing effect* **do**
        if *ContextValue*$(o, oc) < K_3$ and $|$*InFlow*$(o)| > K_4$ **then**
            if *ContextCorrelation*$(oc, k+1)$ **then**
                let $oc = (o_1, \ldots, o_l, \ldots, o_n)$ and $l = min(n, k-n)$
                $depth[o] = k + 1$
                $depth[o_1] = k$
                $depth[o_2] = k - 1$
                $\vdots$
                $depth[o_l] = k - l + 1$
            else
                $depth[o] = 0$
            end if
        else
            $depth[o] = n$
        end if
    end for
    return *depth*
end procedure

This step of determining context depths completes the pre-analysis. We now have a set of pairs $\langle o_i, d_i \rangle$ where d_i represents the depth necessary for the object o_i.

Now the main analysis is straightforward. It applies a selective object-sensitive points-to analysis on the whole input program. This analysis applies a fixed object sensitivity to objects not in the kernel. For every object in the kernel, the context depth as identified by the pre-analysis is applied.

5 Implementation and Experiments

We implemented eclectic points-to analysis (EPA) in DOOP framework [BS09] using the Soufflé Datalog engine [SJSW16]. We used SQLite to compute the metrics of Section 4. We ran our experiments on a Xeon E5-2699 2.30GHz machine having 396 GB RAM with Soufflé using up to 8 cores in parallel.

For our experiments, we considered two programs: OpenJDK7-b147 and Jython 2.1 (called OpenJDK and Jython in the sequel respectively). OpenJDK is one of the largest java program (more precisely a library) available. From [SKB14], Jython is one of the toughest programs in the DaCapo benchmark [BGH+06], to analyze precisely. The size of these programs are given in Table 2. The numbers are in thousands.

Program	Variables	Invocations	Heap Allocations
OpenJDK	1440	312	185
Jython	275	81	28

Table 2: Program Sizes

As JDK is a library, the points-to analysis is extended with the construction of the Most General Application (MGA) [AKS15] modeling the open-world (unknown application) assumptions.

For the purposes of our experiments, we set the default maximum context-sensitivity to be 3O+3H. Note that it is not feasible to compute 3O+3H context-sensitive points-to analysis for either of the OpenJDK and the Jython libraries in a reasonable amount of time.

5.1 Applying the EPA framework

As our EPA technique depends on identifying a suitable kernel, we have to ensure that the identification cost is reasonable. Otherwise, the benefits of a faster points-to analysis is offset by the cost of identifying the kernel.

We experimented with multiple values of thresholds on these programs and selected the best among our experiments. With $K_1 = 20000$ and $K_2 = 50000$,

we extract the kernel from OpenJDK. The extracted kernel for the OpenJDK7 has 92% of the object allocation sites and 70% of the invocations, and takes about 19 minutes to compute. With $K_1 = 10000$ and $K_2 = 10000$ we extract the kernel from Jython. The kernel for Jython has 82% of the object allocations sites and 87% of the invocations, and takes less than 5.5 minutes to compute. We use objects of type Arrays as the candidate objects and set the thresholds $K_3 = 200$ and $K_4 = 200$ for computing the metrics for both OpenJDK and Jython.

5.2 Advantages of EPA

Table 3 contrasts the runtime (minutes) and memory requirements (gigabytes) of EPA for standard fixed 2O+1H object-sensitive and context-insensitive (CI) points-to analysis on the benchmarks. Essentially, it shows that we reduce 27% and 82% of runtime in analyzing OpenJDK and Jython respectively.

Program	CI	2O+1H	EPA
OpenJDK	5.6 min, 6.5 G	270 min, 186 G	198 min, 153 G
Jython	4.2 min, 2.7 G	56 min, 50 G	10 min, 13 G

Table 3: Computational efficiency of EPA

We compare the precision of EPA and 2O+1H using three clients—size of variable to object points-to relation, size of alias relation and size of call graph edges relation . Fig. 4 shows the percentage of tuples (variable x object) removed from the context-insensitive points-to analysis. Fig. 5 shows the percentage of tuples removed from the context-insensitive alias relation—the greater the number, the higher the precision. In regards to size of the relation representing call graph edges, EPA maintains the same precision as 2O+1H analysis. For OpenJDK, both the analyses remove 51.3% of context insensitive call graph edges and for Jython, they remove 98.3% of context insensitive call graph edges. From these clients, it is clear that for a possible slight loss of precision we gain substantial efficiency.

6 Related Work

In this section we discuss the related works on context-sensitive points-to analysis and show the essential difference to our work. These works can be classified into client-independent and client-dependent/demand-driven analyses.

6.1 Client-independent Analyses

Object sensitivity was first introduced by Milanova et al. [MRR05]. They empirically show that object sensitivity is better than call-site sensitivity for object-oriented programs. They also introduce a parametric object sensitivity for targeted context sensitivity. The first kind of parameters specify the depth of object

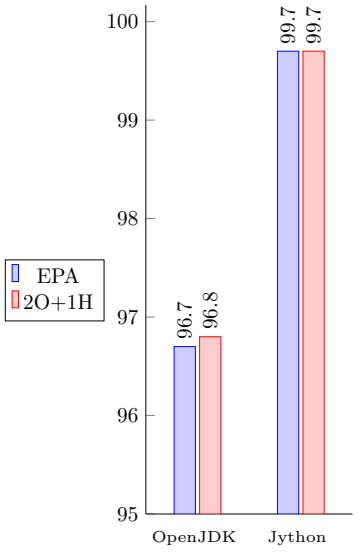

Fig. 4: % of CI Points-to removed

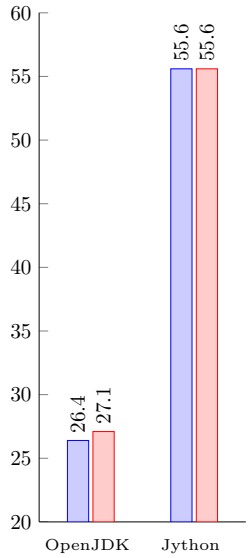

Fig. 5: % of CI Aliases removed

sensitivity for each allocation site. The second kind is on local variables specifying whether they need to be distinguished by the receiver object of the method they are in. Here, the framework user must identify the parts of the program where more or less object sensitivity is required. Our main distinguishing contribution in this paper is to identify these parts of the program and their required context depths.

Smaragdakis et al. [SKB14] proposed an *introspective* context sensitive points-to analysis in DOOP framework. There are two steps in this technique. First, a context-insensitive, Anderson-style points-to analysis is done. Based on metrics such as *PointedToByVar, MethodPointsTo*, they then determine allocation sites and method invocation sites where contexts are necessary. Finally, they perform a fixed object-sensitive analysis switching off context sensitivities at allocation sites and method invocation sites that do not satisfy the selected heuristic. Our experiments on JDK showed that by applying these proposed heuristics on a 2O+1H points-to analysis, the precision did not improve significantly. It removed out only 35% of the context-insensitive points-to facts, whereas EPA removed 96.7%. Hence, we started investigating for techniques that go beyond the binary selection between context-insensitive and a fixed object sensitivity. Thus our technique applies a spectrum of context sensitivities to different parts of the program and achieves scalability without losing precision significantly.

Wei et al. [WR15] propose adaptive context-sensitive points-to analysis for JavaScript programs. Similar to ours, they also do a context-insensitive points-to analysis on the given program as the first stage. Based on its results they

collect characteristics of each function in the program. Then a machine-learning algorithm is used to relate these function characteristics to the kind of context sensitivity to be applied. The four context sensitivities considered are: insensitive, 1 call-site, 1 object, ith-parameter object. Based on the associated context sensitivities for each function, they finally do a selective context sensitive points-to analysis on the whole program. As JavaScript mixes programming paradigms from object-oriented to functional programming style they investigate call-site as well as object-sensitive analyses. Another limitation is that they do not go beyond depth 1 of context sensitivity. For our focus on object-oriented Java programs, we considered only object-sensitive analyses, as has been clearly established as the way to gain precision in Milanova et al. [MRR05] work. We then investigated varied depths of object sensitivity for different allocation sites, which helped us to scale to large programs.

6.2 Client-dependent Analyses

In general, client-dependent analysis suffers from its non-generality, because the technique and the results may not work for a different client. Our objective of points-to analysis is to extract the basic structure of the program, so that many clients such as call-graph construction, taint analysis, escape analysis, make use of it. This objective is the fundamental distinction of the client-dependent context sensitive points-to analysis works.

Oh et al. [OLH+14] use an *impact* pre-analysis to determine at a program point whether call-site context (with predefined depth) is necessary for a method. Here, they deal with C programs, hence the focus is only on method contexts and not heap cloning. First, they fix a call-site context depth. Then they do an *impact* pre-analysis, i.e., they do the context-sensitive (with the defined depth) analysis of the whole program but with a simpler or the simplest abstract domain, such as $(\top, \bot)$. From the results of this analysis, they determine whether to apply context sensitivity (of the defined depth) for a method. This technique is not applicable for our objective of constructing points-to sets for object-oriented programs. This is because finding a very simple abstract domain for points-to analysis for which the context sensitivity can scale to large programs producing relevant information so as to decide the applicability of context sensitivity is not known.

Zhang et al., [ZMG+14] proposed a counter-example guided approach to iteratively clone a method, thus adding call-site sensitivity, for a given set of client queries. Cloning methods based on call-sites is proven to be a unviable way to increase precision for object-oriented programs [MRR05]. Similarly, employing context-insensitive points-to analysis and MAXSAT solver multiple times iteratively is not a feasible method when our objective is to scale large programs of the order of JDK.

Shape analysis [TCR13,GCRN11,SRW99] addresses the problem of analyzing arrays and data structures with the focus on precision. Shape analysis typically analyzes a particular property about data structures (such as cyclicity in lists) and are in general very expensive. Though we have to deal with data structures,

their properties are not central to our objective of making points-to analysis more precise independent of the client. Hence, to scale to large Java programs, works on shape analyses are not directly applicable.

7 Conclusion

In this paper we have presented a framework to scale object-sensitive points-to analysis for large object-oriented programs. It involves identifying and experimenting on the kernel of the program. Then based on our metrics, a selective object-sensitive points-to analysis is applied on the input program. Our experiments on large programs from DaCapo [BGH+06] benchmarks show the effectiveness of our technique.

As part of future work, we would like to investigate machine-learning techniques to train and identify allocation sites in the program where deeper/shallower contexts are useful.

References

[AKS15] Nicholas Allen, Padmanabhan Krishnan, and Bernhard Scholz. Combining type-analysis with points-to analysis for analyzing Java library source-code. In *Proceedings of the SOAP Workshop*, pages 13–18. ACM, 2015.

[And94] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, Fall 1994.

[BGH+06] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khan, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J Eliot B Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA 06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2006.

[BS09] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceeding of the 24th ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 243–262. ACM, 2009.

[GCRN11] Bhargav S. Gulavani, Supratik Chakraborty, G. Ramalingam, and Aditya V. Nori. Bottom-up shape analysis using lisf. *ACM Trans. Program. Lang. Syst.*, 33(5):17:1–17:41, November 2011.

[LLA07] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 278–289. ACM, 2007.

[MRR05] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transaction on Software Engineering Methodolology*, 14(1):1–41, January 2005.

[OLH+14] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. Selective context-sensitivity guided by impact pre-analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 475–484. ACM, 2014.

[SB15a] Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Foundations and Trends in Programming Languages*, 2(1):1–69, 2015.

[SB15b] Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Foundations and Trends in Programming Languages*, 2(1):1–69, April 2015.

[SBL11] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: Understanding object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 17–30. ACM, 2011.

[SGSB05] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 59–76, New York, NY, USA, 2005. ACM.

[SJSW16] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. On fast large-scale program analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction (CC)*, pages 196–206. ACM, 2016.

[SKB14] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. Introspective analysis: Context-sensitivity, across the board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 485–495. ACM, 2014.

[SRW99] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 105–118, New York, NY, USA, 1999. ACM.

[TCR13] Antoine Toubhans, Bor-Yuh Evan Chang, and Xavier Rival. Reduced product combination of abstract domains for shapes. In *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*, pages 375–395, 2013.

[WL04] John Whaley and Monica S. Lam. 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*, PLDI '04, pages 131–144. ACM, 2004.

[WR15] Shiyi Wei and Barbara G. Ryder. Adaptive context-sensitive analysis for javascript. In *29th European Conference on Object-Oriented Programming ECOOP*, volume 37 of *LIPIcs*, pages 712–734. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[ZMG+14] Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. On abstraction refinement for program analyses in Datalog. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*. ACM, 2014.