A Relational Static Semantics for Call Graph Construction

June 19, 2019

Jinan University

Abstract. The problem of resolving virtual method and interface calls in object-oriented languages has been a long standing challenge to the program analysis community. The complexities are brought by various reasons, such as increased levels of class inheritance and polymorphism. In this paper, we propose a new approach called type flow analysis that represent the propagation of type information between program variables by a group of relations without the help of a heap abstraction. We prove that regarding the precision on reachability of class information to a variable, our method produces results equivalent to that one can derive from a points-to analysis. Moreover, in practice, our method consumes lower time and space usage, as supported by the experimental results.

1 Introduction

For object-oriented programming languages, virtual methods (or functions) are those declared in a base class but are meant to be overridden in different child classes. Statically determine a set of methods that may be invoked at a call site is important to program optimization, from result of which a subsequent optimization may reduce the cost of virtual function calls or perform method inlining if target method forms a singleton set, and one may also remove methods that are never called by any call sites, or produce a call graph which can be useful in other optimization processes.

Efficient solutions, such as Class Hierarchy Analysis (CHA) [4, 5] and Rapid Type Analysis (RTA) [3] and Variable Type Analysis (VTA) [12], conservatively assign each variable a set of class definitions, with relatively low precision. Alternatively, with the help of an abstract heap, one may take advantage of points-to analyses [2] to compute a set of object abstractions that a variable may refer to, and resolve the receiver classes in order to find associated methods at call sites.

The algorithms used by CHA, RTA and VTA are conservative, which aim to provide an efficient way to resolve calling edges, and which usually runs linear-time in the size of a program, by focusing on the types that are collected at the receiver of a call site. For instance, let x be a variable of declared class A, then at a call site $x.m()$, CHA will draw a call edge from this call site to method $m()$ of class A and every definition $m()$ of a class that extends A. In case class A does not define $m()$, a call edge to an ancestor class that defines $m()$ will also be included. For a variable x of declared interface I, CHA will draw a call edge from this

```
class A{
    A f;                          1:  A x = new A();  //O_1
    void m(){                     2:  B b = new B();  //O_2
        return this.f;            3:  A y = new A();  //O_3
    }                             4:  C c = new C();  //O_4
}                                 5:  x.f = b;
class B extends A{}               6:  y.f = c;
class C extends A{}               7:  z = x.m();
```

Fig. 1. An example that compares precision on type flow in a program.

Statement	**VTA fact**
$A\ x =$ new $A()$	$x \leftarrow A$
$B\ b =$ new $B()$	$b \leftarrow B$
$A\ y =$ new $A()$	$y \leftarrow A$
$C\ c =$ new $C()$	$c \leftarrow C$
$x.f = b$	$A.f \leftarrow b$
$y.f = c$	$A.f \leftarrow c$
$z = x.m()$	$A.m.this \leftarrow x$ $A.m.return \leftarrow A.f$ $z \leftarrow A.m.return$

Fig. 2. VTA facts on the example

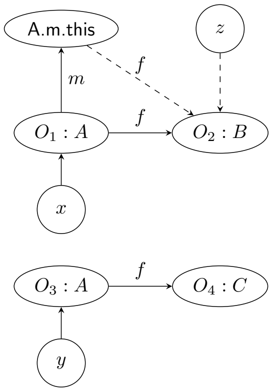

Fig. 3. Points-to results on the example

call site to every method of name $m()$ defined in class X that implements I. We write $CHA(x, m)$ for the set of methods that are connected from call site $x.m()$ as resolved by Class Hierarchy Analysis (CHA). Rapid Type Analysis (RTA) is an improvement from CHA which resolves call site $x.m()$ to $CHA(x, m) \cap inst(P)$, where $inst(P)$ stands for the set of methods of classes that are instantiated in the program.

Variable Type Analysis (VTA) is a further improvement. VTA defines a node for each variable, method, method parameter and field. Class names are treated as values and propagation of such values between variables work in the way of value flow. As shown in Figure 2, the statements on line $1-4$ initialize type information for variables x, y, b and c, and statements on line $5-7$ establish value flow relations. Since both x and y are assigned type A, $x.f$ and $y.f$ are both represented by node $A.f$, thus the set of types reaching $A.f$ is now $\{B, C\}$. (Note this is a more precise result than CHA and RTA which assigns $A.f$ with the set $\{A, B, C\}$.) Since $A.m.this$ refers to x, $this.f$ inside method $A.m()$ now refers to $A.f$. Therefore, through $A.m.return$, z receives $\{B, C\}$ as its final set of types reaching it.

The result of a context-insensitive subset based points-to analysis [2] creates a heap abstraction of four objects (shown on line $1-4$ of Figure 1 as well as in the ellipses in Figure 3). These abstract objects are then inter-connected via field store access defined on line $5-6$. The derived

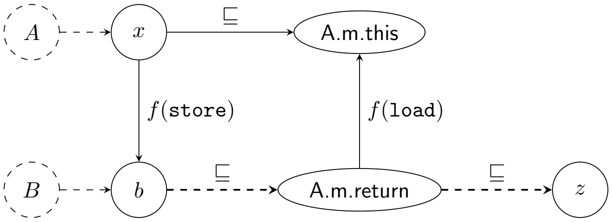

Fig. 4. Type Flow Analysis for variable z in the example

field access from $A.m.this$ to O_2 is shown in dashed arrow. By return of the method call $z = x.m()$, variable z receives O_2 of type B from $A.m.this.f$, which gives a more precise set of reaching type for variable z.

From this example, one may conclude that the imprecision of VTA in comparison with points-to analysis is due to the over abstraction of object types, such that O_1 and O_3, both of type A, are treated under the same type. Nevertheless, points-to analysis requires to construct a heap abstraction, which brings in extra information, especially when we are only interested in the set of reaching types of a variable.

In this paper we introduce a relational static semantics called Type Flow Analysis (TFA) on program variables and field accesses. Different from VTA, besides a binary value flow relation $\sqsubseteq$ on the variable domain $\texttt{VAR}$, where $x \sqsubseteq y$ denotes all types that flow to x also flow to y, we also build a ternary field store relation $\rightarrow \subseteq \texttt{VAR} \times \mathcal{F} \times \texttt{VAR}$ to trace the *load* and *store* relationship between variables via field accesses. This provides us additional ways to enlarge the relations $\sqsubseteq$ as well as $\rightarrow$. Taking the example from Figure 1, we are able to collect the store relation $x \xrightarrow{f} b$ from line 5. Since $x \sqsubseteq \mathsf{A.m.this}$, together with the implicit assignment which loads f of $\mathsf{A.m.return}$, we further derives $b \sqsubseteq \mathsf{A.m.return}$ and $b \sqsubseteq z$ (dashed thick arrows in Figure 4). Therefore, we assign type B to variable z. The complete reasoning pattern is depicted in Figure 4. Nevertheless, one cannot derive $c \sqsubseteq z$ in the same way.

We have proved that in the context insensitive inter-procedural analysis setting, TFA is as precise as subset based points-to analysis regarding type related information. Since points-to analysis can be enhanced with various types of context-sensitivity on variables and objects (e.g., call-site-sensitivity [10, 7], object-sensitivity [8, 11, 13] and type-sensitivity [11]), our type flow analysis will only require to consider contexts on variables, which is left for future work. The context-insensitive type flow analysis has been implemented in the SOOT framework [1], and has been tested on a collection of benchmark programs from specjvm2008 and DaCapo. (Need more citations here, and brief the experimental results.)

2 Type Flow Analysis

We define a core calculus consisting of most of the key object-oriented language features, as shown in Figure 5. A program is defined as a code base $\overline{C}$ (i.e., a collection of class definitions) with statement s to be evaluated. To run a program, one may assume that s is the default (static) entry method with local variable declarations $\overline{D}$, similar to e.g., Java and C++, which may differ in specific language designs. We define a few auxiliary functions. fields maps class names to its fields, $\mathit{methods}$ maps class names to its defined or inherited methods, and type provides types (or class names) for objects. Given class c, if $f \in \mathit{fields}(c)$, then $\mathit{ftype}(c, f)$ is the defined class type of field f in c. Similarly, give an object o, if $f \in \mathit{fields}(\mathit{type}(o))$, then $o.f$ may refer to an object of type $\mathit{ftype}(\mathit{type}(o), f)$, or an object of any of its subclass at runtime. Write $\mathcal{C}$ for the set of classes, $\mathtt{OBJ}$ for the set of objects, $\mathcal{F}$ for the set of fields and $\mathtt{VAR}$ for the set of variables that appear in a program.

$$
\begin{array}{ll}
C & ::= \mathsf{class}\ c\ [\mathsf{extends}\ c]\ \{\overline{F};\ \overline{M}\} \\
F & ::= c\ f \\
D & ::= c\ z \\
M & ::= m(x)\ \{\overline{D}; s; \mathsf{return}\ x'\} \\
s & ::= e \mid x{=}\mathsf{new}\ c \mid x{=}e \mid x.f{=}y \mid s; s \\
e & ::= \mathsf{null} \mid x \mid x.f \mid x.m(y) \\
\mathit{prog} & ::= \overline{C}; \overline{D}; s
\end{array}
$$

Fig. 5. Abstract syntax for the core language.

In this simple language we do not model common types (e.g., int and float) that are irrelevant to our analysis, and we focus on the reference types which form a class hierarchical structure. Similar to Variable Type Analysis (VTA), we assume a context insensitive setting, such that every variable can be uniquely determined by its name together with its enclosing class and methods. For example, if a local variable x is defined in method m of class c, then $c.m.x$ is the unique representation of that variable. Therefore, it is safe to drop the enclosing the class and method name if it is clear from the context. In general, we have the following types of variables in our analysis: (1) local variables, (2) method parameters, (3) this reference of each method, all of which are syntactically bounded by their enclosing methods and classes.

We enrich the variable type analysis with the new type flow analysis by using three relations, a partial order on variables $\sqsubseteq\ \subseteq \mathtt{VAR} \times \mathtt{VAR}$, a type flow relation $\dashrightarrow\ \subseteq \mathcal{C} \times \mathtt{VAR}$, as well as a field access relation $\longrightarrow\ \subseteq \mathtt{VAR} \times \mathcal{F} \times \mathtt{VAR}$, which are initially given as follows.

Definition 1. *(Base Relations) We have the following base facts for the three relations.*

1. $c \dashrightarrow x$ if there is a statement $x = \mathsf{new}\ c$;
2. $y \sqsubseteq x$ if there is a statement $x = y$;

3. $x \xrightarrow{f} y$ *if there is a statement* $x.f = y$*.*

Intuitively, $c \dashrightarrow x$ means variable x may have type c (i.e., c flows to x), $y \sqsubseteq x$ means all types flow to y also flow to x, and $x \xrightarrow{f} y$ means from variable x and field f one may access variable y.[1] These three relations are then extended by the following rules.

Definition 2. *(Extended Relations)*

1. *For all statements* $x = y.f$*, if* $y \xrightarrow{f\ *} z$*, then* $z \sqsubseteq^* x$*.*
2. $c \dashrightarrow^* y$ *if* $c \dashrightarrow y$*, or* $\exists x \in \texttt{VAR} : c \dashrightarrow^* x \wedge x \sqsubseteq^* y$*;*
3. $y \sqsubseteq^* x$ *if* $x = y$ *or* $y \sqsubseteq x$ *or* $\exists z \in \texttt{VAR} : y \sqsubseteq^* z \wedge z \sqsubseteq^* x$*;*
4. $y \xrightarrow{f\ *} z$ *if* $\exists x \in \texttt{VAR} : x \xrightarrow{f} z \wedge (\exists z' \in \texttt{VAR} : z' \sqsubseteq^* y \wedge z' \sqsubseteq^* x)$*;*
5. *The type information is used to resolve each method call* $x = y.m(z)$*.*

$$\begin{array}{l} \forall c \dashrightarrow^* y : \\ m(z')\{\ldots \mathsf{return}\ x'\} \in methods(c) : \end{array} \begin{cases} z \sqsubseteq^* c.m.z' \\ c \dashrightarrow^* c.m.this \\ c, m.x' \sqsubseteq^* x \end{cases}$$

The final relations are the least relations that satisfy constraints of Definition 2. Comparing to VTA [12], we do not have field reference $c.f$ for each class c defined in a program. Instead, we define a relation that connects the two variable names and one field name. (to compare in the motivating example) Although the three relations are inter-dependent, one may find that without method call (i.e., Definition 2.4), one may identify that the two relations $\rightarrow^*$ (field access) and $\sqsubseteq^*$ (variable partial order) can be determined without considering the type flow relation $\dashrightarrow^*$.

In order to compare the precision of TFA with points-to analysis, we present a brief list of the classic subset-based points-to rules for our language in Figure 6, where $param(type(o), m))$, $this(type(o), m))$ and $return(type(o), m)$ refer to the formal parameter, this reference and return variable of the method m of the class for which object o is declared, respectively.

statement	**Points-to constraints**
$x = \mathsf{new}\ c$	$o_i \in \Omega(x)$
$x = y$	$\Omega(y) \subseteq \Omega(x)$
$x = y.f$	$\forall o \in \Omega(y) : \Phi(o, f) \subseteq \Omega(x)$
$x.f = y$	$\forall o \in \Omega(x) : \Omega(y) \subseteq \Phi(o, f)$
$x = y.m(z)$	$\forall o \in \Omega(y) : \begin{cases} \Omega(z) \subseteq \Omega(param(type(o), m)) \\ \Omega(this(type(o), m)) = \{o\} \\ \forall x' \in return(type(o), m) : \\ \qquad \Omega(x') \subseteq \Omega(x) \end{cases}$

Fig. 6. Constraints for points-to analysis.

[1] Note that VTA treats statement $x.f = y$ as follows. For each class c that flows to x which defines field f, VTA assigns all types that flow to y also to $c.f$.

To this end we present the first result of the paper, which basically says type flow analysis has the same precision regarding type based check, such as call site resolution and cast failure check, when comparing with the points-to analysis.

Theorem 1 *In a context-insensitive analysis, for all variables x and classes c, $c \dashrightarrow^{*} x$ iff there exists an object abstraction o of c such that $o \in \mathit{\Omega}(x)$.*

Proof. (sketch) For a proof sketch, first we assume every object creation site $x = \mathsf{new}\ c_i$ at line i defines a mini-type c_i, and if the theorem is satisfied in this setting, a subsequent merging of mini-types into classes will preserve the result.
Moreover, we only need to prove the intraprocedural setting which is the result of Lemma 1. Because if in the intraprocedural setting the two systems have the same smallest model for all methods, then at each call site $x = y.m(a)$ both analyses will assign y the same set of classes and thus resolve the call site to the same set of method definitions, and as a consequence, each method body will be given the same set of extra conditions, thus all methods will have the same initial condition for the next round iteration. Therefore, both inter-procedural systems will eventually stabilize at the same model. □

Lemma 1. *In a context-insensitive intraprocedural analysis where each class c only syntactically appears once in the form of* $\mathsf{new}\ c$*, for all variables x and classes c, $c \dashrightarrow^{*} x$ iff there exists an object abstraction o of type c such that $o \in \mathit{\Omega}(x)$.*

Proof. Since the points-to constraints define the smallest model $(\mathit{\Omega}, \mathit{\Phi})$ with $\mathit{\Omega} : \mathtt{VAR} \to \mathtt{OBJ}$ and $\mathit{\Phi} : \mathtt{OBJ} \times \mathcal{F} \to \mathcal{P}(\mathtt{OBJ})$, and the three relations of type flow analysis also define the smallest model that satisfies Definition 1 and Definition 2, we prove that every model of points-to constraints is also a model of TFA, and vice versa. Then the least model of both systems must be the same, as otherwise it would lead to contradiction.

$(\star)$ For the 'only if' part $(\Rightarrow)$, we define $\mathit{Reaches}(x) = \{c \mid c \dashrightarrow^{*} x\}$, and assume a bijection $\xi : \mathcal{C} \to \mathtt{OBJ}$ that maps each class c to the unique object o that is defined (and $\mathit{type}(o) = c$). Then we construct a function $\mathit{Access} : \mathcal{C} \times \mathcal{F} \to \mathcal{P}(\mathcal{C})$ and show that $(\xi(\mathit{Reaches}), \xi(\mathit{Access}))$ satisfies the points-to constraints. Define $\mathit{Access}(c, f) = \{c' \mid x \xrightarrow{f\ *} y \wedge c \in \mathit{Reaches}(x) \wedge c' \in \mathit{Reaches}(y)\}$. We prove the following cases according to the top four points-to constraints in Figure 6.

- For each statement $x = \mathsf{new}\ c$, we have $\xi(c) \in \xi(\mathit{Reaches}(x))$;
- For each statement $x = y$, we have $\mathit{Reaches}(y) \subseteq \mathit{Reaches}(x)$ and $\xi(\mathit{Reaches}(y)) \subseteq \xi(\mathit{Reaches}(x))$;
- For each statement $x.f = y$, we have $x \xrightarrow{f} y$, then by definition for all $c \in \mathit{Reaches}(x)$, and $c' \in \mathit{Reaches}(y)$, we have $c' \in \mathit{Access}(c, f)$, therefore $\xi(c') \in \xi(\mathit{Reaches}(y))$ we have $\xi(c') \in \xi(\mathit{Access}(\xi(c), f))$.

- For each statement $x = y.f$, let $c \in \mathit{Reaches}(y)$, we need to show $\xi(\mathit{Access}(c, f)) \subseteq \xi(\mathit{Reaches}(x))$, or equivalently, $\mathit{Access}(c, f) \subseteq \mathit{Access}(x)$. Let $c' \in \mathit{Access}(c, f)$, then by definition, there exist z, z' such that $c \in \mathit{Reaches}(z)$, $c' \in \mathit{Reaches}(z')$ and $z \xrightarrow{f\ *} z'$. By $c \in \mathit{Reaches}(y)$ and Definition 2.4, we have $y \xrightarrow{f\ *} z'$. Then by Definition 2.1, $z' \sqsubseteq^* x$. Therefore $c' \in \mathit{Reaches}(x)$.

($\star$) For the 'if' part ($\Leftarrow$), let $(\varOmega, \varPhi)$ be a model that satisfies all the top four constraints defined in Figure 6, and a bijection $\xi : \mathcal{C} \to \mathtt{OBJ}$, we show the following constructed relations satisfy value points-to.

- For all types c and variables x, $c \dashrightarrow^* x$ if $\xi(c) \in \varOmega(x)$;
- For all variables x and y, $x \sqsubseteq^* y$ if $\varOmega(x) \subseteq \varOmega(y)$;
- For all variables x and y, and for all fields f, $x \xrightarrow{f\ *} y$ if for all $o_1, o_2 \in \mathtt{OBJ}$ such that $o_1 \in \varOmega(x)$ and $o_2 \in \varOmega(y)$ then $o_2 \in \varPhi(o_1, f)$.

We check the following cases.

- For each statement $x = \mathsf{new}\ c$, we have $\xi(c) \in \varOmega(x)$, so $c \dashrightarrow^* x$ by definition.
- For each statement $x = y$, we have $\varOmega(y) \subseteq \varOmega(x)$, therefore $y \sqsubseteq^* x$ by definition.
- For each statement $x.f = y$, we have for all $o_1 \in \varOmega(x)$ and $o_2 \in \varOmega(y)$, $o_2 \in \varPhi(o_1, f)$, which derives $x \xrightarrow{f\ *} y$ by definition.
- For each statement $x = y.f$, given $y \xrightarrow{f\ *} z$, we need to show $z \sqsubseteq^* x$. Equivalently, by definition we have for all $o_1 \in \varOmega(y)$ and $o_2 \in \varOmega(z)$, $o_2 \in \varPhi(o_1, f)$. Since points-to relation gives $\varPhi(o_1, f) \subseteq \varOmega(x)$, we have $o_2 \in \varOmega(x)$, which derives $\varOmega(z) \subseteq \varOmega(x)$, the definition of $z \sqsubseteq^* x$.
- The proof for the properties in the rest of Definition 2 are about transitivity of the three TFA relations, which are quite straightforward. We leave them for interested readers. □

3 Implementation and Optimization

We implement the analysis in the SOOT framework [1]. The three base relations ($\dashrightarrow$, $\sqsubseteq$ and $\rightarrow$) of Definition 1 are extracted from program source code and the extended relations ($\dashrightarrow^*$, $\sqsubseteq^*$ and $\rightarrow^*$) of Definition 2 are then computed considering the mutual dependencies between them. Since we are only interested in reference types, we do not carry out analysis on basic types such as boolean, int and double. We also do not consider more advanced Java features such as functional interfaces and lambda expressions, as well as more complex usages including Java Native Interface and reflection based method calls. Array accesses are treated conservatively—all type information that flows to one member of a reference array flows to all members of that array, so that only one node is generated for each array.

Since call graph information may be saved and be used for subsequent analyses, we propose the follow two ways to reduce storage of computed result. If two variables are similar regarding type information in a graph representation, they may be merged and then referred to by the merged node.

1. If $x \sqsubseteq^* y$ and $y \sqsubseteq^* x$, we say x and y form an *alias pair*, written $x \sim y$. Intuitively, $\texttt{VAR}/\sim$ is a partition of $\texttt{VAR}$ such that each $c \in \texttt{VAR}/\sim$ is a strongly connected component (SCC) in the variable graph, which can be quickly collected by e.g., Tarjan's algorithm [14].
2. A more aggressive compression can be achieved in a way similar to bisimulation minimization of finite state systems [6, 9]. Define $\approx \subset \texttt{VAR} \times \texttt{VAR}$ such that $x \approx y$ is symmetric and if
   - for all class c, $c \dashrightarrow^* x$ iff $c \dashrightarrow^* y$, and
   - for all $x \xrightarrow{f\ *} x'$ there exists $y \xrightarrow{f\ *} y'$ and $x' \approx y'$.

It is straightforward to see that $\approx$ is a more aggressive merging scheme.

Lemma 2. *For all $x, y \in \texttt{VAR}$, $x \sim y$ implies $x \approx y$.*

do we need to explain more about the optimization process?

Algorithm 1 split

```
Require: valueToSplit:$set$, splitter:$Value \times Value \times Type \times Field$
Ensure: isSplit:$boolean$, include:$set$, exclude:$set$
  isSplit=false;
  include=$set$();
  exclude=$set$();
  for each v in valueToSplit do
    relations = getRelations(v);
    for each r in relations do
      if r match splitter then
        include.add(v);
        isSplit=true;
        break;
      end if
    end for
  end for
  if isSplit then
    exclude=valueToSplit-include;
  end if
```

4 Experiment

We evaluated our approach by measuring its performance in terms of generated callsites and runtime on 13 benchmark test cases. *Compress*, *crypto* are from the SPECjvm2008 suite, and the rest of those are from the Dacapo suite. We randomly selected these test cases based on its size increasing. All of our experiments were conducted on an Intel i5-8250U processor at 1.60 GHz and 8 GB memory running Ubuntu 16.04LTS, with the Openjdk 1.8.0.

We compared our approach against CHA and PTA implemented by Soot, the most popular static analysis framework for Java. We run context-insensitive PTA because our approach is context-insensitive, thus results will be more comparable. Our evaluation addressed the following three research questions:

$RQ1$: How efficient is our approach compared with the traditional class hierarchy analysis and points-to analysis?
$RQ2$: Is our approach more accurate than CHA or PTA?
$RQ3$: How significantly does our optimazation algorithm achieve to reduce space consumption?

4.1 Implementation

The implemention of our approach is written in Java. We use Soot as our basic framework to extract SSA based representation of the benchmark code. Also, we compared against CHA and PTA implemented by Soot. Our approach optional provides visualization of relations we generated for friendly analysis.

4.2 RQ1:Efficiency

In the first research question, we executed 10 times on each benchmark test case with CHA, PTA and TFA. We calculated runtime consumption on average and each different kind of relation our approach generated. As shown in Table 1, our approach is more efficient than PTA, and generally more efficient than CHA, especially when the program callsite increasing. Time cost in our approach is basically depending on the size of generated relations. More specifically, most of the time is consumed to calculate the fixpoint. For example, CHA, PTA analyze the benchmark *bootstrap* about 23.74 and 34.13 seconds, respectively. TFA only excutes about 0.03 second. The reason is mostly because the relations only reach to 661 in total. There are 3 benchmarks, *xalan, antlr, batik*, can not be analyzed by PTA because it raise the jvm GC overhead limit error. From what we can know, this error is thrown because the jvm garbage collector is taking an excessive amount of time (by default 98% of all CPU time of the process) and recovers very little memory in each run (by default 2% of the heap). We put the label "N/A" in Table 1

4.3 RQ2:Accurancy

We answer the second question with considering generated callsite as the aspect of accurancy. In type flow analysis, a method call will be resolve depending on the type which the caller can access through the relation we generated and that type must have the privage to access the method. Table 3 shows the callsite generated by different analysis. We calculate the origin callsite, the callsite directly written in source code, as the baseline. Our approach reduces a satisfying number of callsite, yielding corresponding accurancy. The result is comparable to what can be achieved by the class hierarchy analysis and the points-to analysis.

Table 1. Time cost with different analysis

bench	$\mathbf{T}_{cha}(s)$	$\mathbf{T}_{pta}(s)$	$\mathbf{T}_{tfa}(s)$	$\mathbf{R}_{type}$	$\mathbf{R}_{\sqsubseteq}$	$\mathbf{R}_{\xrightarrow{f}}$
compress	0.02	0.12	0.02	87	202	24
crypto	0.01	0.12	0.03	94	226	18
bootstrap	23.74	34.13	0.03	191	453	17
commons-codec	0.009	0.12	0.13	306	3324	49
junit	24.45	34.24	0.18	1075	5772	241
commons-httpclient	0.009	0.12	0.36	2423	8511	521
serializer	22.60	32.68	1.71	3006	17726	331
xerces	21.69	34.83	3.74	12590	72503	2779
eclipse	21.95	42.64	1.68	7933	37435	1620
derby	22.77	48.82	18.09	20698	191854	5386
xalan	78.40	N/A	42.11	32971	162249	3696
antlr	44.20	N/A	3.96	16117	76741	3879
batik	45.84	N/A	6.38	29409	122534	6039

Table 2. Optimalization result

bench	$\mathbf{Node}_{origin}$	$\mathbf{Node}_{opt}$	**Reduce**	$\mathbf{Time}(s^{-1})$
compress	205	130	36.59%	0.008
crypto	312	158	49.36%	0.010
bootstrap	514	279	45.72%	0.019
commons-codec	1742	886	49.14%	0.202
junit	5859	3243	44.65%	2.269
commons-httpclient	9708	5164	46.81%	4.651
serializer	9600	6668	30.54%	3.198
xerces	41634	N/A	N/A	N/A
eclipse	34631	N/A	N/A	N/A
derby	112265	N/A	N/A	N/A
xalan	103697	N/A	N/A	N/A
antlr	56589	N/A	N/A	N/A
batik	89336	N/A	N/A	N/A

4.4 RQ3:Optimazation

We use a split algorithm to merge the node that have the same behavior. Thus we can reduce the space consumption. In the third reseach question, we calculated the number of relation nodes before and after optimazation. Besides, time consumption is another factor that we consider. The results are showed in Tabel 2. We evaluated our optimazation algorithm on all benchmarks 10 times and sucessfully exceuted our approach on 7 benchmarks. It shows that we can reduce the space by about 45% on average, with a reasonable time consuming. Another 6 benchmarks can not be execute because the same reason metioned above with PTA. We put the label "N/A" on Table 2.

Table 3. Callsite generated with different analysis

bench	$\mathbf{CS}_{origin}$	$\mathbf{CS}_{cha}$	$\mathbf{CS}_{pta}$	$\mathbf{CS}_{tfa}$
compress	153	160	18	73
crypto	302	307	62	121
bootstrap	657	801	891	328
commons-codec	1162	1372	270	554
junit	3196	17532	11176	1197
commons-httpclient	6817	17118	567	2928
serializer	4782	9533	1248	1756
xerces	24579	56252	10631	8120
eclipse	23607	95073	70016	9379
derby	69537	180428	85212	16381
xalan	57430	155866	N/A	18669
antlr	62007	147014	N/A	17177
batik	56877	235071	N/A	20901

5 Related Work

References

1. Soot. `https://sable.github.io/soot/`. Accessed: 2019-06-10.
2. L. O. Andersen. *Program analysis and specialization for the C programming language.* PhD thesis, DIKU, University of Copenhagen, May 1994.
3. David F. Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, pages 324–341, 1996.
4. Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, pages 77–101, 1995.

5. Mary F. Fernández. Simple and effective link-time optimization of modula-3 programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 103–115, 1995.
6. P. C. Kanellakis and S. A. Smolka. Ccs expressions,
nite state processes, and three problems of equivalence. *Information and Computation*, 86:43—-68, 1990.
7. George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 423–434, 2013.
8. Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, January 2005.
9. R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16:973—-989, 1987.
10. Olin Grigsby Shivers. *Control-flow Analysis of Higher-order Languages or Taming Lambda*. PhD thesis, 1991.
11. 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, 2011.
12. Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 264–280, 2000.
13. Tian Tan, Yue Li, and Jingling Xue. Making k-object-sensitive pointer analysis more precise with still k-limiting. In *International Static Analysis Symposium, SAS'16*, 2016.
14. R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1:146—-160, 1972.