Efficient Points-To Analysis for Partial Call Graph Construction

Zhiyuan Wan\textsuperscript{1}, Bo Zhou\textsuperscript{1}, Ye Wang\textsuperscript{2}, Yuanhong Shen\textsuperscript{1}

\textsuperscript{1}College of Computer Science and Technology, Zhejiang University, China
\textsuperscript{2}School of Computer Science and Information Engineering, Zhejiang Gongshang University, China

Email: \{wanzhiyuan, bzhou\}@zju.edu.cn, yewang@mail.zjgsu.edu.cn, yhshen@zju.edu.cn

Abstract

Many static analysis tools provide whole-program analysis to generate call graphs. However, the whole-program analysis suffers from scalability issue. The increasing size of the libraries exacerbates the issue. For many Web applications, the libraries (e.g. Servlet containers) are even not available for whole-program analysis. We present HyPta, a points-to analysis approach, to construct partial call graphs for Java programs. HyPta extends the standard points-to analysis by establishing a hybrid heap model. Since our approach does not analyze the method bodies of the library classes, the heap model distinguishes between the abstract memory locations in the application and those in the library. HyPta infers the pointer information in the library from the interactions between the application and the library. We implement HyPta based on Spark framework and evaluate it on 14 widely cited Java benchmarks. The evaluation shows that HyPta is faster than Averroes and Spark by a factor of 4.9x and 13.7x, respectively. Meanwhile, it constructs sound and precise partial call graphs.

1. Introduction

A call graph is a crucial prerequisite for most inter-procedural static analyses used in compilers, verification tools and program understanding tools \[6\]. Many established static analysis tools and frameworks provide whole-program analysis to generate call graphs. However, for modern object-oriented programs, scalability of the whole-program analysis has been a significant hurdle for its practical use in real-world tools. The increasing size of the libraries exacerbates the scalability issue. The libraries include (1) standard libraries (e.g. Java J2SE or C++ STL), (2) domain-specific libraries (e.g. graphics and linear algebra), and (3) extensible frameworks and middleware. The code of the library accounts for a large amount of the code in a program \[13\]. Moreover, for many Web applications, the libraries (e.g. Servlet containers) are not available for analysis since they depend on the deployment environment. Therefore, partial call graph construction is more practical and scalable than whole-program analysis.

To construct sound and precise partial call graphs, we propose an efficient points-to analysis approach named HyPta. HyPta analyzes the application classes, as well as the signatures, the fields and the class hierarchy of the library classes referenced by the application. Since the memory locations in the library are unknown to our approach, HyPta extends the standard points-to analysis by establishing a hybrid heap model. The heap model distinguishes between the abstract memory locations in the application and those in the library. This leads to a hybrid points-to set for each variable in the program. HyPta also analyzes the interactions between the application and the library to infer the abstract objects created in the library. The hybrid points-to set for the library is used to resolve the library call-back edges and restrict the abstract objects flow back to the application part. We implement HyPta based on the Soot \[11\] framework and evaluates it on 14 real-world Java programs. The experimental results show that HyPta reduces the execution time of Spark and Averroes by a factor of 13.7x and 4.9x, respectively. Meanwhile, HyPta generates sound and precise call graphs of the application part.

2. Proposed Approach

2.1. Overview

The points-to analysis is a subset-based, flow- and context-insensitive, and field-sensitive points-to analysis. It builds the call graphs on the fly. The input is a set of Java classes designated as the application classes. The application classes may depend on other Java classes designated as the library classes. The analysis only analyzes the method signatures, the fields and the class hierarchy of the library
classes that are referenced by the application classes. Since the memory locations in the library is unknown, we propose a hybrid heap model to distinguish between the memory locations in the application and those in the library. We use allocation sites to represent the abstract objects created in the application and use class names to represent those created in the library. The hybrid heap model leads to a hybrid points-to set for each variable, which consists of two parts: the application points-to set and the library points-to set.

**Definition 1** (Application Points-to Set). For each pointer variable $v$ in the program, the application points-to set is defined as $Pt_{A}(v)$, which contains abstract objects that are created in the application and represented by their allocation sites.

**Definition 2** (Library Points-to Set). For each pointer variable $v$ in the program, the library points-to set is defined as $Pt_{L}(v)$, which contains abstract objects that are created in the library and represented by their class names.

**Sets and Notations.** The analyzed program is composed of a set of classes $Cls$. The classes define a set of methods $Method$. The classes declared in the application are denoted by $Cls_{A}$ while those declared in the library are denoted by $Cls_{L}$. Each class $cls \in Cls$ has a set of fields $fields_{cls} = \{f_0, f_1, ..., f_{q-1}\}$. Some of the fields are non-static and denoted by a set $F$ while some are static and denoted by a set $SF$. The static fields can be viewed as some sort of global variables. An artificial field $[]$ is introduced to model array cells. Each method $m \in Method$ has a list of parameters, a set of local variables and a body. The set of variables in the application is represented by $V$. If a method $m$ has $k$ parameters, the parameters are in order $p_0, p_1, ..., p_{k-1}$. For non-static methods, $p_0$ is this parameter. In the Java programming language, parameters are a special case of local variables, i.e., $p_i \in V$. Let $O$ be the set of all abstract objects created in the program. We denote by $O_A$ the set of all the abstract objects created in the application and $O_L$ the set of those created in the library. The body of the method $m$ contains a list of statements from the set $Statement$. For brevity, we discuss only a set of elementary statements that manipulate pointers shown in Table 1. $v_i \in V$ denotes a local variable or a formal parameter (including this) in the program. Since our analysis is flow-insensitive, there is no need to consider control-flow statements such as branches and loops. CALL and RETURN statements handle the interprocedural control-flow. Each CALL statement corresponds to a call site. We denote by $C$ the set of call sites in the application classes.

### 2.2. Program Representation

Our points-to analysis builds a pointer assignment graph (PAG) [7] to represent the program being analyzed. The PAG consists of three types of nodes: allocation nodes, variable nodes and field dereference nodes. Allocation nodes represent allocation sites and model the heap locations in the application. Variable nodes represent local variables, method parameters, return values and static fields. Field dereference nodes represent filed access expressions and are parameterized by corresponding variable nodes that are dereferenced by the field access. Note that the allocation nodes in the PAG form the set of abstract objects $O_A$. The nodes in the PAG are connected with four types of edges reflecting the pointer flow: new/newarray, assign, store, and load.

We construct the PAG by associating elementary statements with different PAG entities. Table 1 shows the different elementary statements and corresponding PAG entities. Note that $v, v_i, v_R, C.f$ and $this$ denote the variable nodes, $o$ denotes the allocation node, and $v_i.f$ denotes field dereference nodes. In the handling of interprocedural flow, for each method $m$, the PAG has a node for each of $m$'s formal parameters and a special $ret_{m}$ node for $m$'s return value. At each call site $c$ of $m$, we add an assign edge from each actual parameter to its formal parameter, and an assign edge from the $ret_{m}$ node to the caller’s variable. Each node $n$ in the PAG is associated with a points-to set, which contains the abstract objects that may be referenced by the node. The abstract objects in the points-to sets are propagated along the PAG edges.

### 2.3. Library Model

Since our analysis does not take into account the method bodies of the library classes, we propose a model to represent the library as below:
Variables. Let Library be an arbitrary variable in the library.

Methods. The analysis is defined in terms of mL for an arbitrary method in the library.

Points-to set. A single points-to set Pt(Library) is constructed for the library, which consists of two parts. One is PtL(Library), which contains o ∈ O that is referenced by the library. Since we assume that the library methods can create and reference the objects of any classes declared in the library, PtL(Library) contains the class names of all library classes implicitly. The other is PtA(Library), which contains o ∈ OA that is referenced by the library.

Call sites. cL represents an arbitrary call site in the library and can invoke: 1) any visible method declared in any library class; 2) the method m declared in an application class cls, only if m is non-static and overrides a method originally declared in a library class, and there exists o ∈ PtA(Library) where StaticType(o) is cls or a subclass of cls. We use the notation StaticType(o) for the static type of object o.

The interaction between the application and the library determines the abstract objects that may escape from the application scope to the library, and vice versa. We infer the set of abstract objects OL created in the library by analyzing the interaction. Furthermore, the library points-to sets of the variables are updated according to the sequence of rules that correspond to the types of elementary statements as below.

Rule 1. for v1 = v2.f, where the instance field f is declared in cls ∈ ClsL:
StaticType(f) ∈ OL; StaticType(f) ∈ PtL(v1).

The notation StaticType(o) denotes the static type of the variable o.

Rule 2. for v = X.f, where X ∈ ClsL:
StaticType(f) ∈ OL; StaticType(f) ∈ PtL(v).

Consider a call to a static method m declared in a library class; the call occurs in an application class.

Rule 3. for w = X.m(v1, ..., vk), where X ∈ ClsL:
StaticType(retm) ∈ OL; StaticType(retm) ∈ PtL(w).

In this rule, v1 is the argument of the call site. We use retm to denote the return value of X.m.

Consider an instance call w = v0.m(v1, ..., vk) occurring in a reachable method in the application, where m is a method declared in the library.

Rule 4. for w = v0,m(v1, ..., vk), where v0 ∈ V:
StaticType(retm) ∈ OL; StaticType(retm) ∈ PtL(w).

Consider an instance call occurring in the library with the target method m, which is also the target of a library call back edge. The method is originally declared in the library and overridden by the application.

Rule 5. for Library,m(v1, ..., vk):
StaticType(this) ∈ OL; StaticType(this) ∈ PtL(this);
StaticType(vi) ∈ OL; StaticType(vi) ∈ PtL(vi) ∧ 1 ≤ i ≤ k.

2.4. Virtual Call Handling

To soundly and precisely handle virtual calls, we compute targets of virtual calls and construct the call graphs on the fly, i.e., as the relevant points-to sets of call site receivers are computed.

Definition 3 (CallGraph). We denote by CallGraph the partial call graph of the application part. The CallGraph is comprised of a finite set of call edges. Each call edge connects a call site, which is a statement in some method, to a method that may be invoked from that call site. The call edge is represented by a 3-tuple (m1,m2,c) where m1,m2 ∈ {mL} ∪ Method and c ∈ [cls] ∪ C.

Consider a virtual call v.m(...) occurring in the method M declared in the application, the analysis resolves the call site c ∈ C on the receiver v by using the points-to set PtA(v) and PtL(v) according to the following two rules:

Rule 6. for each object o ∈ PtA(v), S taticLookup (cls, m) = m′ where cls = StaticType(o) : (M,m′,c) ∈ CallGraph.

Rule 7. for each object cls ∈ PtL(v), StaticLookup (cls,m) = m′ : (M,m′,c) ∈ CallGraph.

We use the notation StaticType(o) to denote the static type of the object o, and the notation StaticLookup(cls,m) to denote the definition (if any) of a method with name m that one finds when starting a static method lookup in the class cls.

For the virtual calls in the library, we define the following rule to compute library call back edges with PtA(Library):

Rule 8. for each object o ∈ PtA(Library), m is declared in an application class cls where cls ∈ StaticLookup(o) ∪ Supertypes(StaticType(o)) and overrides a library method: (mL,m′,cL) ∈ CallGraph.

Supertypes(cls) denotes the supertypes of cls.

3. Evaluation

We evaluate HyPta by comparing its performance and generated call graphs with those of Spark and Averroes. Spark [7] implements the whole-program analysis to construct call graphs. Averroes [2] generates a placeholder library to enable Spark and other whole-program analysis frameworks to construct partial call graphs.1

Experimental Setting. The experiment is executed on a machine with Intel Core 2 2.13GHz p7450 CPU and 2 GB of RAM. We ran the experiment on the DaCapo benchmark program v. 2006-10-MR2 [5] and the SPEC jvm 98 benchmark program [8]. We use the same settings as earlier published work [1, 2]. The experiment analyzed the Java standard library from jre 1.4.2,11. The jython benchmark has been excluded because the incompleteness of referenced classes. The jython benchmark is also

1See http://plg.uwaterloo.ca/~karim/projects/averroes.
excluded because sophisticated reflection forms are heavily used, which makes static analysis impractical. The detailed information of the benchmarks are presented in Table 2.

**Implementation.** We implemented HyPta on top of the Soot framework [11], version 2.5.0, and bootstrapped by Spark [7]. We defined the application part of a program as the analysis scope for Soot. We extend Spark by creating the hybrid points-to sets, propagating the objects in the hybrid points-to sets and resolving call sites using the points-to sets according to different rules.

### 3.1. Performance

Figure 1 shows the running time required for call graph construction by HyPta, Spark and Averroes. HyPta scales well taking under 32 seconds for each benchmark. We found HyPta is faster than Averroes by a factor of 4.9x on average (min: 1.9x, max: 33.9x, geometric mean: 4.9x) and faster than Spark by a factor of 13.7x on average (min: 3.3x, max: 131.2x, geometric mean: 13.7x). HyPta improves the performance of Spark significantly. This improvement is achieved by generating call graphs without analyzing the method bodies of the library classes. The running time for Averroes breaks into two components: pre-analysis time (denoted by $\text{Averroes}^{\text{placeholder}}$ in Figure 1) is the time required for Averroes to generate placeholder library; and analysis time (denoted by $\text{Averroes}^{\text{Analysis}}$ in Figure 1) is the time required for Spark to generate call graphs with the Averroes placeholder library. The slowdown of Averroes is due to the generation of the placeholder library.

### 3.2. Call Graph Soundness

We evaluated the soundness of the static call graphs by counting the call edges that are present in the dynamic call graphs collected by $^\ast$J [4] but missing from the static call graphs generated by static tools. The dynamic call graphs are observed during the dynamic execution of the benchmarks. The results are shown in Table 3. The row “Dynamic” shows the number of call edges in the application part of the benchmarks. The rows “Dynamic\HyPta”, “Dynamic\Averroes” and “Dynamic\Spark” show that the number of dynamic call edges that are missing in the static call graphs generated by HyPta, Spark and Averroes. The rows “Dynamic\HyPta” and “Dynamic\Averroes” show only two library call graph edges in $\text{lusearch}$ and $\text{xalan}$ are missing from the call graphs of all the benchmarks generated by HyPta and Averroes. In $\text{lusearch}$, a $\text{NullPointerException}$ is thrown at runtime and a call to the constructor of this exception is recorded in the dynamic call graph. In $\text{xalan}$, a call to $\text{java.lang.ref.Finalizer.register}$ is missing. The reason is that both HyPta and Averroes do not model the complete runtime behavior of the Java virtual machine. Furthermore, the row “Dynamic\Spark” shows that Spark is missing a significant number of call edges for the benchmarks that make heavy use of reflection. This is because Spark does not handle reflection while HyPta and Averroes make use of the information of reflection collected by TamiFlex [3].

### 3.3. Call Graph Precision

We compared the number of call edges in the call graphs generated by HyPta, Averroes and Spark with respect to three categories of call edges. Spark implements a whole-program analysis call graph construction algorithm that analyzes the library thoroughly, whereas HyPta and Averroes construct partial call graphs without analyzing the method bodies of the library classes. Therefore, we say that HyPta or Averroes is precise when the generated call graphs are identical to those generated by Spark. In addition, since we found some dynamic call edges are missing from the call graphs generated by HyPta, Averroes and Spark, we add the missing edges to the static call graphs first to avoid confounding due to the differences in soundness. The rows “HyPta\Spark” and “Averroes\Spark” in Table 4, 5 and 6 represent the numbers of call edges in the static call graphs generated by HyPta and Averroes that are missing in the call graphs generated by Spark, and are not present in the dynamic call graphs.

**Application Call Graph Edges.** Table 4 shows that HyPta generates precise call graphs with respect to application call graph edges for $\text{lusearch}$, $\text{compress}$, $\text{db}$, $\text{jess}$ and $\text{raytrace}$. For all benchmarks, HyPta generates an average of 343 and a median of 22 extra application call graph edges (min: 0, max: 1821, average: 343, median: 22) in comparison with Spark. The median is negligible as it represents 0.49% of the median number of application call graph edges generated by Spark. The $\text{bloat}$, $\text{hsqldb}$ and $\text{xalan}$ benchmarks have the highest frequency of imprecise call edges. This is due to the fact that these benchmarks have large subsystems that implement the library interfaces, $\text{java.util.}$, JDBC and $\text{XML}$, respectively. Moreover, the average is smaller than

### Table 2: Information about the benchmarks.

<table>
<thead>
<tr>
<th>Program</th>
<th># Application Classes</th>
<th># Application Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>antlr</td>
<td>202</td>
<td>1330</td>
</tr>
<tr>
<td>bloat</td>
<td>334</td>
<td>2526</td>
</tr>
<tr>
<td>chart</td>
<td>489</td>
<td>1042</td>
</tr>
<tr>
<td>hsqldb</td>
<td>389</td>
<td>3213</td>
</tr>
<tr>
<td>luindex</td>
<td>324</td>
<td>501</td>
</tr>
<tr>
<td>lusearch</td>
<td>324</td>
<td>896</td>
</tr>
<tr>
<td>pmd</td>
<td>527</td>
<td>1936</td>
</tr>
<tr>
<td>xalan</td>
<td>534</td>
<td>3107</td>
</tr>
<tr>
<td>compress</td>
<td>12</td>
<td>32</td>
</tr>
<tr>
<td>db</td>
<td>3</td>
<td>31</td>
</tr>
<tr>
<td>jack</td>
<td>56</td>
<td>292</td>
</tr>
<tr>
<td>javac</td>
<td>176</td>
<td>1105</td>
</tr>
<tr>
<td>jess</td>
<td>5</td>
<td>18</td>
</tr>
<tr>
<td>raytrace</td>
<td>25</td>
<td>159</td>
</tr>
</tbody>
</table>
Figure 1: Comparing the time taken by the analysis in HyPta, Averroes and Spark for each benchmark program.

Table 3: Comparing the soundness of HyPta, Averroes and Spark.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>HyPta</th>
<th>Averroes</th>
<th>Spark</th>
</tr>
</thead>
<tbody>
<tr>
<td>antlr</td>
<td>3449</td>
<td>4257</td>
<td></td>
</tr>
<tr>
<td>bloat</td>
<td>657</td>
<td>1627</td>
<td></td>
</tr>
<tr>
<td>chart</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hsqldb</td>
<td>726</td>
<td>539</td>
<td></td>
</tr>
<tr>
<td>luindex</td>
<td>2087</td>
<td>2953</td>
<td></td>
</tr>
<tr>
<td>lusearch</td>
<td>43</td>
<td>54</td>
<td></td>
</tr>
<tr>
<td>pmd</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xalan</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>compress</td>
<td>596</td>
<td>2538</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jack</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>javac</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jess</td>
<td>13</td>
<td>330</td>
<td></td>
</tr>
<tr>
<td>raytrace</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

the average of 414 and the median is slightly smaller than the median of 23 (min: 0, max: 2310, average: 414, median: 23) for Averroes, which suggests HyPta generates more precise call graphs than Averroes with respect to application call graph edges.

Library Call Graph Edges. Table 5 shows that HyPta generates precise call graphs with respect to library call graph edges for antlr, luindex, compress and raytrace. For all benchmarks, HyPta generates an average of 20 and a median of 2 extra library call graph edges (min: 0, max: 116, average: 20, median: 2) in comparison with Spark. The median is negligible as it represents 0.55% of the median number of library call graph edges generated by Spark. Since the average is smaller than the average of 22 and the median is smaller than the median of 3 (min: 0, max: 116, average: 22, median: 2.5) for Averroes, HyPta generates more precise call graph than Averroes with respect to library call graph edges.

Library Call Back Edges. Table 6 shows the results of comparing the library call back edges. For all benchmarks, HyPta generates an average of 85 and a median of 23 extra application call back edges (min: 1, max: 645, average: 85, median: 22.5) in comparison with Spark. The average is larger than the average of 69 and the median is larger than the median of 5 (min: 0, max: 605, average: 69, median: 4.5) for Averroes. The median represents 31.69% of the median number of application call graph edges generated by Spark. We investigate the causes of imprecise library call back edges generated by HyPta. One cause is that HyPta models the implicit calls to ⟨clinit⟩ as library call back edges under 3 conditions: a) when a static method is invoked, b) when a static field is accessed, and c) when an object or an array of objects is initialized. Whereas Spark models the implicit calls to ⟨clinit⟩ as application call graph edges. Therefore, most of these extra library call back edges are with the static initialization blocks ⟨clinit⟩ as their targets. Another cause is that HyPta implements partial call graph construction algorithm that makes assumption about the objects referenced in the library. HyPta assumes the objects passed to the library through calls to library methods would be accessible to the library. However, some of the library methods may not leak a reference to other library code. This assumption in HyPta introduces imprecision into the points-to sets inside the library and results in spurious library call back edges. Furthermore, these objects may pollute the points-to sets of the variables in the application and result in imprecise call edges of other categories.

The results show that HyPta achieves higher precision than Averroes, especially on bloat and xalan benchmarks, with respect to the application call graph edges. The target methods of the extra call edges are declared in a relatively small number of application classes. These classes implement the interfaces or extend the abstract classes declared in the library. For instance, for bloat, the targets of the extra edges are declared in the classes edu.purdue.cs.bloat.util.ImmutableIterator and edu.purdue.cs.bloat.codegen.Liveness$1, which implement the library interface java.util.Iterator. This is due to the construction process of library points-to set, which determines the number of targets when resolving call sites.

4. Related Work

We discuss the most related work that designs efficient approaches to construct partial call graphs. The Wala framework [12] ignores the effect of the library code. This
leads to incomplete call graphs because the pointer information is not complete both in the application and in the library. Yan et al.’s work [13] creates procedure summaries for library methods and uses them when analyzing a specific client program. However, the creation and use of summary face a number of technical challenges in terms of analysis abstraction, algorithms and infrastructure support. Tip and Palsberg [10] associate a single points-to set \( S_E \) with the the library and incorporate refinements to make less conservative assumptions about the library. Rountev et al. [9] create a placeholder, a main method, to represent the unanalyzed part. The main method contains a variety of placeholder statements to represent all possible behaviors of the unanalyzed code. This may alter the original behaviors of the application code and result in imprecise call graphs of the application part. Averroes [2] generates a placeholder library and enables the whole-program analysis frameworks to construct partial call graphs efficiently. It builds on the separate compilation assumption proposed in Ali and Lhotá’s previous work [1]. The placeholder library implements the specific constraints derived from the separate compilation assumption that overapproximates the behaviors of the library.

5. Conclusions and Future Work

We propose an efficient points-to analysis approach to construct partial call graphs for Java applications. Our approach 1) uses a hybrid heap model to distinguish between the abstract memory locations in the application and those in the library; 2) models the interaction between the application and the library to infer the abstract objects created in the library. We implement the analysis based on the Spark framework and evaluated the analysis by comparing it with Spark and Averroes. The results show that our approach provides an efficient way to construct sound and precise partial call graphs of the application part. In the future, we plan to extend the work to handle reflection and static initialization blocks more precisely.

References