Side-Effect Analysis with Fast Escape Filter

Binxian Tao, Ju Qian
College of Computer Science and Technology
Nanjing University of Aeronautics and Astronautics
Nanjing 210016, China
{binxiantao, jqian}@nuaa.edu.cn

Xiaoyu Zhou
School of Computer Science and Engineering
Southeast University
Nanjing 210096, China
zhouxy@seu.edu.cn

ABSTRACT
Side-effect analysis is a fundamental static analysis used to determine the memory locations modified or used by each program entity. For the programs with pointers, the analysis can be very imprecise. To improve the precision of side-effect analysis, many approaches design more elaborate background pointer analyses in order to obtain smaller side-effect sets, but very few approaches consider to achieve better precisions by refining the side-effect analysis algorithms themselves. To address the problem, this paper presents a new side-effect analysis approach that uses Gay and Steensgaard’s fast escape analysis to filter superfluous side-effects. The approach does not need to modify the background pointer analysis and can filter side-effects in the intraprocedural level and the interprocedural level. The experimental results show that it can effectively improve the analysis precision within a short extra time.

Categories and Subject Descriptors
D.3.4 [Programming Languages]: Processors – Compilers.

General Terms
Algorithms, Experimentation, Languages.

Keywords
Side-effect analysis, escape analysis, points-to analysis

1. INTRODUCTION
Side-effect analysis is a fundamental static analysis used by many software engineering tools to determine the memory locations modified or used by each program entity (e.g., method). For Java programs, it highly depends on pointer information in order to resolve the indirect memory accesses. The pointers also make the side-effect analysis difficult to be precise enough. To improve the precision of side-effect analysis, many approaches design more elaborate background pointer analyses [15] in order to obtain smaller side-effect sets [9, 16, 19, 24, 26, 30], but very few approaches consider to achieve better precisions by refining the side-effect analysis algorithms themselves. We believe improving the side-effect analysis algorithms is also very important, because it can add extra precision to the analysis and focusing on the side-effect analysis algorithms themselves also can let the pointer analysis and side-effect analysis be evolved independently.

For the improving of the side-effect analysis algorithms, in our previous work [23], we have used a lazy access path resolving technique to successfully improve the precision of side-effect analysis under an inclusion-based context-insensitive points-to analysis. Besides this approach, some other approaches [25, 27] also use escape analysis, a technique to determine whether an object can escape from its creation scopes, to improve the precision of side-effect analysis. However, they only use object-based escape analyses (e.g., [5, 6, 8]) which tag escape information to heap objects to refine side-effect analysis. No research has considered to use the more efficient variable-based escape analyses (e.g., [3, 14]), which tag escape information to local variables instead of heap objects, in side-effect computation, and no research has presented any experimental data to show how much improvement can be achieved with escape analysis. To address these problems, this paper proposes an approach that incorporates Gay and Steensgaard’s fast variable-based escape analysis [14] in side-effect computation. The approach can use escape information to filter superfluous side-effects during intraprocedural side-effect collection and interprocedural side-effect propagation and thereby improve the side-effect analysis precision.

In the paper, we focus on the method level side-effect analysis problem. The approach firstly uses Spark inclusion-based context-insensitive points-to analysis [18] to compute the pointer information and uses Gay and Steensgaard’s escape analysis to determine whether the objects accessed by each reference variable are local to the enclosing method. Then, collecting the intraprocedural side-effects and propagating the side-effects from the callee methods to the caller methods can obtain the final side-effects for each method. During the collection of each method’s intraprocedural side-effects, all the object fields modified or used via the reference variables only accessing local objects will be filtered out from the side-effect sets. When propagating a callee’s side-effects to a caller, a key issue is to filter out the modification or use effects on the objects that escape from the callee but do not escape from the caller. To filter out such superfluous side-effects, we represent a callee’s side-effects as access paths starting from formal parameters, if possible. Then, such access paths can be mapped to the ones starting from actual arguments in the caller, and we can use the variable-based escape information in the caller to filter out the superfluous parts of these side-effects.

To validate the proposed approach, we conduct an experiment on several popular benchmark programs. The results show that the side-effect analysis with escape filter can effectively improve the analysis precision within a short extra time, especially when the class initialization calls and the finalize calls are ignored and the immutable types like java.lang.Integer, java.lang.Float, and java.lang.String are treated as built-in types. In the latter case, the newly proposed approach can achieve about 25.3% precision improvement for the total side-effects, in which the analysis precision for the modification effects can averagely be improved by 34.3% and the analysis precision for the use effects can averagely be improved by 14.4%.

The rest of the paper is organized as follows. Section 2 presents a motivating example to show the problems in side-effect analysis. Section 3 introduces the newly proposed side-effect analysis method. Section 4 is the experimental study. Finally, we discuss the related work and conclude the paper.
2. A MOTIVATING EXAMPLE

This section presents a motivating example to show why we need escape information in side-effect analysis and how the escape information can be used.

In Figure 1, suppose the objects created by statements 1, 5, and 7 are modeled into abstract objects O1, O2, and O3, respectively. Then, with points-to analysis, we will find that the location accessed by access path b.i is O1.i, while the locations accessed by access path c.i could be O2.i or O3.i. Without knowing whether an object’s lifetime exceeds its creation method, it is very difficult to determine whether an object is visible outside a method. Therefore, in a normal side-effect analysis, all the reads or writes to O1, O2, and O3 will be considered having out-visible use or modification effects. For method foo, we will obtain a modification effect set \{O1.i, O2.i, O3.i\} and a use effect set \{O1.i\}. For method bar, its side-effects include the ones in method foo. Because there is no intraprocedural modification or use effect in bar, bar’s side-effects are just the same as that of method foo. According to such side-effect sets, we will find that there are define-use relationships between statements 8 and 9. Such relationships will forbid some compiler optimizations that need to switch the order between these two statements.

In fact, the above side-effect sets are safe but over-conservative. In the program of Figure 1, the lifetime of object O1 does not escape from method foo, and the lifetime of object O2 does not escape from method bar. For method foo, object O1 is not visible outside the method, and hence its modification or use effects do not contain O2.i. For method bar, since all the objects accessed inside it are not out-visible, its side-effect sets are actually empty. With these side-effect sets, we can find that there is no real define-use relationship between statements 8 and 9. The two statements do not read or write the same locations, and hence their order can be safely switched.

To avoid the superfluous side-effects due to non-out-visible locations, this paper firstly uses escape analysis to identify whether the objects accessed by each local variable are local to the enclosing method. Then, if a modification or use is performed via a local variable that only accesses local objects, the modification or use will be filtered out during side-effect collection. For example, in Figure 1, with an escape analysis, we can find that variable b in method foo only accesses local objects. Therefore, the modifications or uses via variable b will be discarded. In method foo, these access paths will be put into the caller’s side-effect sets. Otherwise, they will be discarded. In method bar, with escape analysis, we can find that variable d only accesses local objects. Therefore, the caller access path d.i will not be put into the modification effect set of method bar, and bar’s side-effect sets are empty. By filtering the side-effects of the callees in the callers, we can obtain more precise side-effects for the caller methods.

### Figure 1. A motivating example

```java
class C {
    int i;
    5 C d = new C(); // O2
    6 } foo(C a){
    7 void zar(){
        1 C b = new C(); // O1
        2 b.i = 1;
        3 C c = a;
        4 c.i = b.i;
    }
}

void foo(C a){
    void zar();
    1 C b = new C(); // O1
    2 b.i = 1;
    3 C c = a;
    4 c.i = b.i;
}
```

3. SIDE-EFFECT ANALYSIS WITH ESCAPE FILTER

This section firstly introduces the escape analysis used in the paper. Then, we present the concept of interstatement must alias. With such alias, the modified or used access paths inside a method can be mapped to the access paths on formal parameters at the method entry. By representing the side-effects as entry access paths, we can filter a callee’s side-effects in the callers. Finally, we present the detail of the side-effect analysis algorithm which has an escape filter to avoid superfluous side-effects.

3.1 Escape Analysis

This paper uses Gay and Steensgaard’s fast algorithm [14] to do escape analysis. The algorithm establishes three predicates, i.e., fresh, escaped, and returned, which can be used in a combination to determine whether a local reference variable may access out-visible objects. Each predicate is a map from local variables to Boolean values:

- **fresh**: Local → Boolean
- **escaped**: Local → Boolean
- **returned**: Local → Boolean

Here, fresh(v) = true indicates that variable v only accesses objects newly created by the enclosing method of v. escaped(v) = true indicates that variable v accesses an object possibly escaping from the enclosing method via static fields or instance fields. returned(v) = true indicates that variable v accesses an object possibly escaping from the enclosing method via the method returns.

Based on Gay and Steensgaard’s predicates, we define a predicate isLocal(v) to determine whether a variable v can only access the objects whose lifetimes are limited to the enclosing method of v.

\[
\text{isLocal(v)} := \text{fresh(v)} \land \neg \text{escaped(v)} \land \neg \text{returned(v)}
\]

Predicate isLocal(v) is evaluated to true only when variable v just accesses newly created objects and the objects do not escape from the enclosing method of v via static fields, instance fields, or method returns. If isLocal(v) = true, then any modification or use to the fields of the objects accessed by v will not be put into the side-effect sets.

In methods foo and bar of Figure 1, fresh(b) and fresh(d) are true, escaped(b) and escaped(d) are false, and no object escapes via method returns. Therefore, isLocal(b) and isLocal(d) are true. While isLocal(a) and isLocal(C) are false because fresh(a) and fresh(C) are not true. According to the values of the isLocal predicates, we can filter the modifications or uses via variables b or d out of the side-effect sets during the side-effect computation.
3.2 Interstatement Must Alias Analysis

In this paper, we use the forward interstatement must alias to map the inner access paths to the access paths on formal parameters at the method entry. An interstatement must alias models the relation between access paths in different program sites [22]. The definition of the forward must alias is presented as following:

**Definition 1** (Forward Must Alias). A forward must alias \( \langle m, \alpha, \beta \rangle \) indicates that when the program flow is from statement \( m \) to statement \( n \), access path \( \alpha \) in the latest occurrence of \( m \) always accesses the same physical location as access path \( \beta \) in the current occurrence of \( n \).

For example, in Figure 1, \( \langle E_{foo}, a, i, 4; c, i \rangle \) is a forward must alias because from the view of statement 4, access path \( a \) in the latest occurrence of \( E_{foo} \)’s entry, always accesses the same memory location as access path \( c \). According to this forward must alias, we can represent the modification effects on \( c,i \) as access path \( a \). These two representations have the same meaning for side-effect computation.

There can be different algorithms for interstatement must alias computation. This paper chooses the global value numbering [10] based algorithm presented in our previous work [23] to obtain the forward must aliases. The algorithm works on the SSA representation of a program. It is context-insensitive and can determine whether two instance field references or array element references definitely access the same locations in a very short time.

3.3 The Side-Effect Analysis Algorithm

As the computation of the modification effects and the computation of the use effects are almost the same, this section will only present the algorithm for modification effect computation. We focus on the method level side-effect analysis. During the analysis, the side-effects of a method \( m \) are represented with a pair \( \langle A_m, L_m \rangle \), where \( A_m \) is a set of access paths on \( m \)’s formal parameters and \( L_m \) is a set of abstract locations storing the side-effects that cannot be represented by set \( A_m \). After analysis, resolving the memory locations accessed by the access paths in \( A_m \) and unify them with \( L_m \) can obtain the final location-based side-effects.

The side-effect analysis of a method takes two steps: first, we collect the intraprocedural side-effects caused by the method itself; then, we merge the side-effects of the callee methods into the current method to obtain the total side-effects. Algorithm 1 presents the detailed algorithm. The first and the second output for loops correspond to the first and second steps, respectively. In the first step, for each assignment, we firstly check whether the modified access path can access an out-visible location with the isLocal predicate. Note that the side-effect analysis is performed on a three-address code like intermediate program representation. Each access path contains at most one field or array element access. Therefore, if the base variable only refers to local objects, the modified location is definitely not out-visible and should be filtered out from the side-effect sets. Then, for an access path that accesses out-visible locations, we try to map the inner access path to an entry access path on some formal parameter with the interstatement must aliases. If an aliased map target is found, then the target access path will be added to set \( A_m \). Otherwise, the access path will be resolved to memory locations and put into \( L_m \).

In the second step, we merge the side-effects of the callee methods to the callers. Let \( m \) be the caller and \( Q \) be the callee. Then, \( L_Q \), the abstract location part of \( Q \)’s side-effects, will directly be merged into \( L_m \), the modified abstract location set of method \( m \). For the access path form side-effects in \( A_Q \), we firstly map them to the access paths on \( Q \)’s actual arguments. Then, these access paths can be treated as normal modified access paths. We can use the isLocal predicate to filter out the local objects, and put the remaining side-effects into either \( A_m \) or \( L_m \) depending on whether the modified access paths can be mapped to some must aliased entry access paths.

Since the side-effects of a caller depend on the side-effects of its callees, the methods in a program are analyzed in bottom-up manner. When there are recursions, we iteratively compute the side-effect sets until a fixed point is reached. To be efficient, the algorithm follows the strongly-connected-component (SCC) based computing order [11] and restricts the iterative computation to the SCCs in a call graph. In this way, unnecessary iterations can be avoided.

In Figure 1, for method foo, there are two modified access paths: \( b,i \) and \( c,i \). Because isLocal(b) is true, \( b,i \) will be filtered out during the first step of the intra-procedural side-effect collection, and it will not be added to the modification effect set of method foo. Since isLocal(c) is false, \( c,i \) will not be filtered out, and we will try to map \( c,i \) to an entry access path. According to a forward must alias \( \langle E_{foo}, a,i, 4; c,i \rangle \), \( c,i \) can be mapped to entry access path \( a,i \). The access path \( a,i \) will then be put into \( A_{foo} \), the access path part of method foo’s modification effect set. As all the access paths accessing out-visible locations can be mapped to the entry access paths, \( L_{foo} \), the location part of method foo’s modification effect set will be empty.

For method bar, since the method has no intraprocedural side-effect, we only need to merge the side-effects of callee method foo into bar’s side-effect sets. The only side-effect of foo, namely \( a,i \), will firstly be mapped to access path \( d,i \) at the callsite of foo, i.e., statement 6. Then, as isLocal(d) is true, the side-effect represented by \( d,i \) will be filtered out. No element will be put into method bar’s side-effect sets, and the method is side-effect free.

**Algorithm 1** Side-Effect Analysis

<table>
<thead>
<tr>
<th>Input:</th>
<th>m: method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>( \langle A_m, L_m \rangle )</td>
</tr>
</tbody>
</table>

- let \( E_{foo} \) be the entry of \( m \) and \( S \) be the set of all assignments in \( m \);
- let \( Alias(m) \) be the interstatement must aliases in \( m \);
- let \( C \) be the set of all method calls in \( m \);
- let \( Loc(a) \) be the abstract locations accessed by access path \( a \);
- let baseVar(a) be the base variable of access path \( a \);

- foreach \( n: a: \ldots \in S \) do
  - if \( \neg isLocal(baseVar(a)) \) then
    - if \( \exists i. <E_{foo}, b,i; a: \ldots \in Alias(m) \) then
      - \( A_m := A_m \cup \{ b \} \);
    - else
      - \( L_m := L_m \cup Loc(a) \);
  - end
- end

- foreach \( c \in C \) do
  - foreach callsite \( Q \) of \( c \) do
    - <\( A_Q, L_Q \)> := SideEffect(Q);
    - \( L_m := L_m \cup L_Q \);
  - foreach \( a \in A_Q \) do
    - if access path on actuals mapped from \( a \) in \( Q \) then
      - if \( \neg isLocal(baseVar(a)) \) then
        - if \( \exists i. <E_{foo}, b,i; c: \ldots \in Alias(m) \) then
          - \( A_m := A_m \cup \{ b \} \);
        - else
          - \( L_m := L_m \cup Loc(c) \);
      - end
    - end
- end
- end
In the algorithm, we use the \textit{isLocal} predicate based escape filter to check the properties of local variables and thereby filter out the modification or use effects on local objects. As the number of local variables in a method is usually much smaller than the number of abstract objects possibly referred to by them, this filter is usually more efficient than an object-based filter that checks the property of each object to filter superfluous side-effects. The algorithm also incorporates the idea of lazy access path resolving [23] in side-effect analysis, since the accessed locations of many access paths are resolved later in the caller methods instead of in the methods modifying or using the access paths. This also adds another source for the precision improvement of the side-effect analysis.

4. EXPERIMENT

To validate the proposed side-effect analysis method, we implement Gay and Steensgaard’s fast escape analysis, the GVN-based interstatement must alias analysis, and the new side-effect analysis algorithm on Soot 2.4.0 [28] with Spark [18] inclusion-based context-insensitive points-to analysis as the base. The analysis is performed on Soot’s Simple SSA intermediate representation. In the points-to analysis, we name objects according to their allocation sites. Each instance field is counted as a memory location, and after analysis all the access path based side-effect representations are resolved to their accessed locations. As a prototype, the implementation does not consider the side-effects of native methods, but such methods do be simulated during pointer analysis.

4.1 Experiment Settings

Based on the implementation, the paper randomly selects several benchmark programs used in the previous researches on pointer analysis or side-effect analysis for experimental study. As presented in Table 1, the programs include \textit{soot-c} from the Ashes suite [1], \textit{polyglot} 1.3.5 [21], \textit{ps} from DaCapo benchmark suite version beta-2006-08 [4], \textit{213x_javac} and \textit{202_jess} from SPECjvm2008 [2] (two programs used as the inputs of program \textit{compresso}), and another 6 programs from DaCapo benchmark suite version 2006-10-MR2-xdeps[4]. There are also other programs in the above benchmark suites. The experimental results for them are similar to the ones in Table 1. For space limitation, we will not list them here. In Table 1, column \textit{Classes} counts the classes analyzed by Soot. Column \textit{Methods} counts the reachable methods in Spark’s call graph. These methods are processed by size-effect analysis. Column \textit{Spark Time} shows the time consumed by Spark points-to analysis.

In the experiment, we compare the newly proposed side-effect analysis which has an escape filter (\textit{Esc}) with a normal strongly-connected-component based side-effect analysis (\textit{Norm}) [11]. The normal analysis considers all the methods in a SCC of the call graph having the same side-effects. It firstly finds the SCCs of the call graph. Then, merging the side-effects of all the methods in a SCC and their callees can find the side-effects for each method in the SCC. In the paper, all the experiment results are collected on a PC with Intel Core i5 2.8 GHz CPU, 16G RAM, and JDK 1.5.0.09.

4.2 Experimental Results and Discussions

Table 2 presents the first experimental results. In the table, column \textit{Time} lists the time spent in the whole side-effect analysis phase, including the time spent in the escape analysis and the must alias analysis, but not including the time spent in bytecode parsing and points-to analysis. Column \textit{Mod Locations} and \textit{Use Locations} count the sizes of modification sets and the sizes of use sets, respectively. From the table, we can see that for all benchmarks, the new analysis method averagely reduces about 11.5% of a method’s side-effect sets. The analysis time is averagely 5.8 times of the normal algorithm. Although much more than the old one, we shall note that the usual analysis time is still in 1 minute, and the increased analysis time is not too much compared to the total time spent in both the side-effect analysis and the preprocessing like bytecode parsing and points-to analysis before side-effect analysis. A major reason for that the new algorithm takes more time is because it needs fixed point iteration on the strongly-connected-components of the call graph, while the normal algorithm does not demand any iterative computation on recursive loops [11].

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Classes</th>
<th>Methods</th>
<th>Spark Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>soot_c</td>
<td>A bytecode analysis framework for Java</td>
<td>2754</td>
<td>10309</td>
<td>19.2</td>
</tr>
<tr>
<td>polyglot</td>
<td>A framework for Java language extensions</td>
<td>2327</td>
<td>8195</td>
<td>20.2</td>
</tr>
<tr>
<td>antlr</td>
<td>A parser and lexical analyzer generator</td>
<td>2213</td>
<td>8762</td>
<td>11.9</td>
</tr>
<tr>
<td>python</td>
<td>A Python interpreter</td>
<td>2651</td>
<td>11645</td>
<td>16.1</td>
</tr>
<tr>
<td>blast</td>
<td>A Java bytecode optimizer</td>
<td>2380</td>
<td>10737</td>
<td>18.4</td>
</tr>
<tr>
<td>pmd</td>
<td>A Java source code analyzer</td>
<td>2869</td>
<td>10038</td>
<td>11.8</td>
</tr>
<tr>
<td>ps</td>
<td>A postscript interpreter</td>
<td>5104</td>
<td>13378</td>
<td>52.8</td>
</tr>
<tr>
<td>fop</td>
<td>An output-independent print formatter</td>
<td>6625</td>
<td>12704</td>
<td>50.6</td>
</tr>
<tr>
<td>213x_javac</td>
<td>JDK Java compiler</td>
<td>2250</td>
<td>9162</td>
<td>14.8</td>
</tr>
<tr>
<td>202_jess</td>
<td>A Java expert shell system</td>
<td>5071</td>
<td>12606</td>
<td>44.8</td>
</tr>
</tbody>
</table>

Table 1. Experiment subjects

<table>
<thead>
<tr>
<th>Subject</th>
<th>Time (s)</th>
<th>Mod Locations</th>
<th>Use Locations</th>
<th>Improvement (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>soot_c</td>
<td>15.2</td>
<td>58.3</td>
<td>10390</td>
<td>8975</td>
</tr>
<tr>
<td>polyglot</td>
<td>2.6</td>
<td>20.2</td>
<td>10487</td>
<td>9393</td>
</tr>
<tr>
<td>antlr</td>
<td>7.0</td>
<td>37.2</td>
<td>10930</td>
<td>9418</td>
</tr>
<tr>
<td>python</td>
<td>5.6</td>
<td>49.2</td>
<td>14670</td>
<td>13317</td>
</tr>
<tr>
<td>blast</td>
<td>23.8</td>
<td>97.4</td>
<td>13781</td>
<td>11697</td>
</tr>
<tr>
<td>pmd</td>
<td>8.4</td>
<td>31.1</td>
<td>11465</td>
<td>10101</td>
</tr>
<tr>
<td>ps</td>
<td>31.9</td>
<td>117.0</td>
<td>16185</td>
<td>14003</td>
</tr>
<tr>
<td>fop</td>
<td>13.3</td>
<td>97.4</td>
<td>16328</td>
<td>14519</td>
</tr>
<tr>
<td>213x_javac</td>
<td>8.6</td>
<td>36.9</td>
<td>11969</td>
<td>10312</td>
</tr>
<tr>
<td>202_jess</td>
<td>12.4</td>
<td>113.0</td>
<td>16266</td>
<td>14627</td>
</tr>
</tbody>
</table>

Average: 15.8

<table>
<thead>
<tr>
<th>Subject</th>
<th>Time (s)</th>
<th>Mod Locations</th>
<th>Use Locations</th>
<th>Improvement (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>soot_c</td>
<td>15.2</td>
<td>58.3</td>
<td>10390</td>
<td>8975</td>
</tr>
<tr>
<td>polyglot</td>
<td>2.6</td>
<td>20.2</td>
<td>10487</td>
<td>9393</td>
</tr>
<tr>
<td>antlr</td>
<td>7.0</td>
<td>37.2</td>
<td>10930</td>
<td>9418</td>
</tr>
<tr>
<td>python</td>
<td>5.6</td>
<td>49.2</td>
<td>14670</td>
<td>13317</td>
</tr>
<tr>
<td>blast</td>
<td>23.8</td>
<td>97.4</td>
<td>13781</td>
<td>11697</td>
</tr>
<tr>
<td>pmd</td>
<td>8.4</td>
<td>31.1</td>
<td>11465</td>
<td>10101</td>
</tr>
<tr>
<td>ps</td>
<td>31.9</td>
<td>117.0</td>
<td>16185</td>
<td>14003</td>
</tr>
<tr>
<td>fop</td>
<td>13.3</td>
<td>97.4</td>
<td>16328</td>
<td>14519</td>
</tr>
<tr>
<td>213x_javac</td>
<td>8.6</td>
<td>36.9</td>
<td>11969</td>
<td>10312</td>
</tr>
<tr>
<td>202_jess</td>
<td>12.4</td>
<td>113.0</td>
<td>16266</td>
<td>14627</td>
</tr>
</tbody>
</table>

Average: 15.8

Table 2. Experimental results for the normal and the newly proposed side-effect analysis methods
In the previous work [23], we have found that a great threat to the precision improvement is a couple of huge side-effect sets (mainly belonging to the Java library methods) which are hard to refine but widely propagated due to the over-conservative call graph. In practice, to suffer less from them, we can ignore the implicit class initialization calls and finalize calls and treat the immutable types including java.lang.Object, java.lang.String, java.lang.Class, and the wrapper classes corresponding to char, int, float, and so on, as build-in types [23]. These treatments are reasonable because ignoring the class initialization calls and the finalize calls usually does not affect the understanding of a method’s behavior and the objects of immutable types are values instead of variable memory locations. Such treatments make the analysis less safe, but may still benefit many applications like program understanding and software maintenance.

Table 3 presents the experimental results for the side-effect analyses with special treatments to the class initialization calls, the finalize calls, and the immutable types. From this table, we can see that with the treatments presented in the last paragraph the newly proposed algorithm averagely reduces 25.3% of each method’s side-effects. The precision improvement for the modification effects is more significant. The new analysis averagely gets 34.3% precision improvement for the modification effect computation. Although demanding 6.4 times of the analysis time, the average side-effect analysis time is just 40.9 seconds. These results indicate that the new side-effect analysis method can effectively improve the analysis precision within a short extra time, especially for the modification effect analysis.

As mentioned in Section 3.3, the newly proposed approach also incorporates the idea of lazy access path resolving (LAPR) in side-effect analysis. To investigate whether the new approach adds more precision to the LAPR based side-effect analysis [23], we also compare it with the LAPR based side-effect analysis in experiment. Table 4 presents the experimental comparison results. In the table, Column Time, Mod, and Use list the consumed time, the modification effect set size, and the use effect size of the LAPR based side-effect analysis, respectively. Column Improvement shows the precision improvements of the newly proposed side-effect analysis compared to the LAPR based side-effect analysis in modification effect computation, use effect computation, and total side-effect computation. From the table, we can see that the new analysis algorithm improves the side-effect analysis precision by 6.7% compared to the LAPR based algorithm. Their analysis time is very close. This means that using the escape filter can add more precision to the LAPR based side-effect analysis with very little extra cost.

5. RELATED WORK

There is a large body of work on side-effect analysis for languages with pointers. Clausen[9] proposes a field-based approach to compute side-effects for Java programs. The approach does not use any pointer information, and the instances of a field in different objects are not distinguished. Ryder et al. [26] present a general framework for modification side-effect analysis of C programs. They use alias analysis as a base for side-effect computation. Rafzalmahefa[24] presents algorithms for side-effect analysis for Java based on context-insensitive type-based analysis and refers-to analysis. Milanova et al. [19] propose a side-effect analysis for Java based on object-sensitive points-to analysis. Le et al. [16] compute the side-effects for Java programs based on Spark context-insensitive points-to analysis and use the side-effect information in compiler optimization. Xue et al. [30] propose a method to compute side-effects for incomplete Java programs. Qian et al. [23] introduce a lazy access path resolving based method to improve the side-effect analysis precision under inclusion-based context-insensitive pointer analysis. Flexeder et al. [13] propose a method to compute side-effects for assembly code. All the above approaches do not use escape information in side-effect computation. Rountev [25] and Šalcianu and Rinard [27]’s approaches check whether an object is reachable from the outside of a method to determine whether the object escapes, and thereby use the escape information to filter superfluous side-effects. These approaches use the object-based escape information. As discussed before, such information is more complex to be used compared to
the variable-based escape information used by this paper. What’s more, both the two researches do not investigate to what extent the escape information can be used to improve the side-effect analysis precision. Cherem and Rugina [7] propose an approach to build special escape and effect summaries for each method. They do not compute the detailed side-effect sets. Therefore, the approach is not comparable with ours.

Many escape analyses with flavors in different precisions and efficiency have been proposed in literature [3, 5-8, 12, 14, 17, 20, 29]. Some of the analyses are context-insensitive (e.g., [6, 14]), while some others are context-sensitive (e.g., [5, 29]). Several escape analyses can even compute flow-sensitive escape information (e.g., [8, 20, 29]). Most of the existing escape analyses are object-based (e.g., [5, 6, 8, 20, 29]), which tag escape information on heap objects to indicate whether an object can possibly escape from its creation scopes. There are also some variable-based escape analyses which tag escape information on local reference variables instead of heap objects to indicate whether an object can escape (e.g., [3, 14]). These analyses are usually less precise but much more efficient. This paper only investigates the effects of Gay and Steensgaard’s lightweight context-insensitive variable-based escape analysis on side-effect computation. The work cannot comprehensively show the power of escape analysis on side-effect computation. With a more expensive escape analysis, more precision improvement may be achieved. In the future, we also plan to study other escape analysis methods to improve side-effect computation.

6. CONCLUSION

People believe that escape information can be used to improve the precision of side-effect analysis. But, how can the escape information be used and how much improvement can be achieved? These questions have not been comprehensively answered. To address the problems, this paper designs a new side-effect analysis using Gay and Steensgaard’s fast escape analysis to filter superfluous side-effects. The experimental study shows that the new analysis with escape filter can effectively improve the side-effect analysis precision within a short extra time. This indicates that the escape information does be valuable in side-effect computation.

ACKNOWLEDGEMENT

This research is supported by the National Natural Science Foundation of China (60903026).

REFERENCES