Improving Side-Effect Analysis with Lazy Access Path Resolving

Ju Qian  
College of Information Science & Technology  
Nanjing University of Aeronautics and Astronautics  
Nanjing 210016, China  
jqian@nuaa.edu.cn

Yuming Zhou and Baowen Xu  
State Key Lab. for Novel Software Technology  
Nanjing University  
Nanjing 210093, China  
{zhouyuming, bwxu}@nju.edu.cn

Abstract

For scalability, many side-effect analysis methods choose inclusion-based context-insensitive (IBCI) pointer analysis as their basis. However, such a pointer analysis is known to be imprecise, which often results in over-conservative side-effect sets. In this paper, we present a lightweight approach that exploits lazy access path resolving to improve the precision of side-effect analysis under IBCI pointer analysis. The approach partly represents and propagates side-effects in the access path form with the help of interstatement must aliases. All access paths can finally be resolved to the accessed locations, but during the side-effect propagation phase, an access path will never be resolved as long as it could be mapped to another access path in the caller. Since in inclusion-based points-to analysis, points-to sets of variables in the callers tend to be smaller than the ones in the callees, such lazy resolving mechanism can lead to more precision. The experimental results show that the lazy access path resolving approach is effective in improving the precision of IBCI pointer analysis based side-effect analysis methods.

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. For Java programs, it highly depends on pointer information in order to resolve indirect memory accesses. The pointer information can be provided by pointer analyses, which fall into two categories: context-sensitive and context-insensitive [9]. It is widely acknowledged that context-sensitive pointer analyses have difficulties to be scalable. Therefore, many side-effect analyses choose context-insensitive pointer analyses as their bases [10, 16, 22]. However, such pointer analyses are known to be imprecise, which often results in over-conservative side-effect sets. Due to the superfluous data, a side-effect set can be huge. Our experience in Soot compiler infrastructure [21] shows that even for a small method, there can be tens of thousands of abstract locations in its modification set, given that the objects are named by their allocation sites and each field of an abstract object is counted as an abstract location. These huge side-effect sets may complicate the client applications like program slicing and etc.

To provide better support for the client applications, there is a large demand for improving side-effect analysis precision without affecting scalability. Besides, it is often preferred not to redesign the background pointer analysis algorithm, since the pointer analysis may be shared to achieve other goals in the client application and building a separate pointer analysis merely for side-effect collecting may introduce redundant computation. Briefly, we want a lightweight approach which can refine side-effect analysis on the ground of a specified context-insensitive pointer analysis. There are already many side-effect analysis approaches in literature [6, 10, 12, 16–18, 20, 22]. However, as we known, rarely any one meets our requirement. Most of them just exploit precision improvements in the background pointer analyses.

Inspired by the fact that in inclusion-based context-insensitive points-to analysis [3], the points-to sets of variables in the callers tend to be smaller than the ones in the callees, this paper presents a lightweight approach exploiting lazy access path resolving to improve the precision of side-effect analysis under IBCI pointer analysis. For clarity, we only concern on the method-level side-effect analysis. In the new approach, the side-effects of a method are partly represented as access paths (e.g., p.x, p.y) on formal parameters with the help of interstatement must aliases. These side-effects will be propagated from the callees to the callers. During the bottom-up phase, a modified or used access path will never be resolved to the accessed locations as long as it could be mapped to an access path in the caller. Since access paths in the caller can often be resolved to smaller abstract location sets, this lazy resolving mechanism may achieve more precision for side-effect computation.

In the side-effect analysis, interstatement must alias plays a critical role in mapping an access path in the body of a method to an access path on a formal parameter at the entry of the method. It specifies the alias relation between access paths in two program sites, and thereby a modification or use inside a method can be safely modeled as an access
path on the method interface. To collect such aliases, we design two algorithms, both based on global value numbering (GVN) [13], a compiler technique to determine equal values. One is based on a simple intraprocedural GVN which does not trace the value of heap accesses, while the other is based on a more complex interprocedural GVN designed for alias computation. They provide different trade-offs between precision and complexity.

To validate the proposed approach, an empirical study is conducted on several benchmark programs. The results show that the lazy access path resolving is effective in improving side-effect analysis precision. The improvement is particularly significant when class initializations and finalize calls in a method are ignored and the references to immutable classes like java.lang.Integer and java.lang.String are treated as build-in types. In that case, the soundness requirement needs to be slightly relaxed. But now the new approach can averagely get 16.7% more precision for each method in modification and use side-effect analysis (over 26% more precision for modification side-effect computation). Over 25% of methods have their side-effect sets reduced by more than a half, and over 25% of methods can have all their read/write effects represented as access paths. This indicates the proposed analysis would be more beneficial in many applications like program understanding and software maintenance where safety is not always critical.

The experiments also show that the side-effect analysis based on the new interprocedural GVN does not achieve much more precision than that based on a simple intraprocedural GVN which does not trace the value of heap accesses. This means a simple GVN would be enough in side-effect computation. The side-effect analysis incorporating such GVN can improve the analysis precision at an affordable cost, and our new approach shares the same bases and interfaces as common context-insensitive pointer analysis based side-effect computing systems. This makes it very easy to be integrated into many existing program analysis tools.

The rest of the paper is organized as follows. Section 2 introduces the inclusion-based context-insensitive pointer analysis concerned by this paper. Section 3 presents the proposed side-effect analysis method. Section 4 provides two GVN-based interstatement must alias analysis methods. Section 5 is the empirical study. Finally, we discuss the related work and conclude the paper.

2 Inclusion-Based Context-Insensitive Pointer Analysis

Context-insensitive pointer analyses are often unification-based [19] or inclusion-based [3]. Compared to the former approaches, the later ones can produce more accurate results, and in this paper we mainly concern on the inclusion-based points-to analyses with objects distinguished by their allocation sites. Such analyses generally consist of two steps.

Firstly, the program is abstracted into an inclusion constraint system. Then, the points-to set of each pointer is calculated by solving these constraints.

Formally, let $O_s$ denote the objects allocated in an instruction $s$ and $T(p)$ denote the abstract objects pointed to by a reference typed abstract location $p$. Then, the program can be abstracted into 5 kinds of inclusion constraints each corresponding to a basic instruction type. Table 1 illustrates the points-to abstraction.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t = \text{new } C$</td>
<td>object creation</td>
<td>$T(t) \supseteq O_s$</td>
</tr>
<tr>
<td>$t = r$</td>
<td>copy</td>
<td>$T(t) \supseteq T(r)$</td>
</tr>
<tr>
<td>$t.f = r$</td>
<td>store</td>
<td>$T(a.f) \supseteq T(r), a \in T(t)$</td>
</tr>
<tr>
<td>$t = r.f$</td>
<td>load</td>
<td>$T(t) \supseteq T(a.f), a \in T(r)$</td>
</tr>
<tr>
<td>$t = a_0, m(a_1, \ldots, a_k)$</td>
<td>method call</td>
<td>$T(t) \supseteq T(a_0), T(p_0) \supseteq T(a_i)$</td>
</tr>
</tbody>
</table>

Table 1. Constraints on Points-to Sets

The constraint system in Table 1 conservatively assumes that, for an assignment, the target locations always had larger points-to sets than the source locations. In the last row, $r_m$ represents the return value of method $m$. $p_i$ represents the $i$-th formal parameter, in which $p_0$ represents the this pointer corresponding to receiver object $a_0$. Note that for simple illustration, the method call in Table 1 is a call to actual target $m$ instead of a virtual call. For any virtual method call, we should firstly find its possible targets and then build constraints according to the 6th row of Table 1.

In the IBCI pointer analysis, the points-to set of a formal parameter subsumes that of all its corresponding actual arguments. Since a method is often called in many places, the points-to sets of variables in the callers tend to be much smaller than the ones in the callees.

3 Improving Side-Effect Analysis with Lazy Access Path Resolving

This section firstly illustrates the basic idea of our new side-effect analysis with a motivating example. Then, we introduce the interstatement must alias, which forms a foundation for the lazy access path resolving, and show how the side-effects of each method are computed in detail.

3.1 A Motivating Example

Consider the example in Figure 1. Since class Integer is extensively used and for each call $r.intValue()$ there is an inclusion constraint $T(this) \supseteq T(r)$, variable this in method intValue() has a huge points-to set. Similar situations also occur in constructor Integer(int). With such points-to sets, both intValue() and Integer(int) will have huge modification/use sets, and propagating these sets up to the caller will result in huge side-effect sets for method foo().

However, in the execution context of foo(), both method intValue() and constructor Integer(int) can only access objects allocated in foo(). Let the objects allocated in $S1$ and $S2$ be abstracted into $O_{S1}$ and $O_{S2}$, respectively. The
int foo()
{ class C { Integer i; }
    Integer a = new Integer(1);
    Integer b = new Integer(2);
    int i = this.value;
    int ib = b.intValue();
    this.value = i;
    return i;
}

int intValue()
{ return this.value;
}

int i = this.value;

Figure 1. A Motivating Example for the New Side-Effect Analysis

side-effect sets of method foo() can only contain the elements in \{O_{S1}.value, O_{S2}.value\}.

To make the side-effect analysis more precise, we need to figure out that intValue() and Integer(int) can only access objects \{O_{S1}, O_{S2}\}. Unfortunately, this would be very difficult for the traditional context-insensitive points-to and side-effects analysis. One possible solution is using the locations reachable from foo(), i.e., the ones reachable from local variables a and b, to filter the side-effect sets of intValue() and Integer(int) since a and b only access \{O_{S1}, O_{S2}\}. However, in the real case, determining locations reachable from a method is very time-consuming, and this approach is generally not practical.

A more practical solution would be representing the side-effects of a method with the access paths on its formal parameters. When merging the side-effects of a callee to its callers, the access paths in the callee can be mapped to the ones in the callers. Since the callers tend to have smaller points-to sets, this may result in smaller side-effect sets. We call the above approach lazy access path resolving, because a modified or used access path is preferred to be resolved later in the callers. For example in Figure 1, both the use set of method intValue() and the modification set of constructor Integer(int) can be represented as access path set \{this.value\}. In method foo(), such set can further be mapped to \{a.value, b.value\} and lazily resolved. By resolving the accessed locations of \{a.value, b.value\}, we will find much smaller side-effect sets for foo().

Actually, Figure 1(a) just shows a very simple case. The situation can be more complex. Figure 1(b) presents another example. In this case, instance fields are not directly loaded or stored with formal parameters, which makes it difficult to represent side-effect sets as access paths. In method Zar(C), instance field C.i is loaded and stored from a non-parameter variable x. Access path x.i is further copied to variable y, and y.intValue stores another instance field. To represent read/write effects with access paths as much as possible, it is desirable to identify the relation between x.i at \{S5, S6\} and c.i at the method entry, and the relation between y.intValue at S7 and c.i.intValue at the method entry. Such relations can be detected by must alias analysis. The next section presents the definition of interstatement must alias. Then in later sections, we will formalize the new side-effect analysis method, and show how the complex cases are handled.

3.2 Interstatement Must Alias: The Definition

As illustrated in Figure 1(b), objects passed in by method parameters can be accessed with other variables. To represent side-effects with access paths as much as possible, it is desirable to determine whether an access path in the method body definitely accesses the same location as some other one at the method entry. Such relationship is a kind of interstatement must alias in nature. This paper formalizes these aliases into the following definition.

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

\[
\text{void zar(C c)}
\]

\[
\text{forward must alias}
\]

\[
\text{C x = c;}
\]

\[
\text{S5: Integer y = x.i; \{Entry}_{zar}: c.i, S5: x.i\}}
\]

\[
\text{S6: x.i = null; \{Entry}_{zar}: c.i, S6: x.i\}}
\]

\[
\text{S7: y.intValue = 1; \{Entry}_{zar}: c.i.value, S7: y.value\}}
\]

Figure 2. Interstatement Must Aliases

Forward must alias is a special kind of interstatement must alias [15]. Here the word forward means these aliases can be identified by forward tracing. Figure 2 presents the forward must aliases occurring in method Zar(C) of Figure 1(b). Let \( \text{Entry}_{zar} \) denote the entry of Zar(C). In this example, \( \text{Entry}_{zar}: c.i, S5: x.i \) is a forward must alias because when the program flow is from \( \text{Entry}_{zar} \) to \( S5 \), c.i in the latest occurrence of \( \text{Entry}_{zar} \) always accesses the same memory location as \( x.i \) in the current occurrence of \( S5 \). According to the definition, \( \text{Entry}_{zar}: c.i, S6: x.i \) and \( \text{Entry}_{zar}: c.i.value, S7: y.value \) are also two forward must aliases.

With the forward must aliases, a memory access via an access path \( α \) at a program site \( m \) can be easily mapped to an access path \( β \) at the method entry, given there is a forward must alias \( (Entry: β, m: α) \). As the entry has only one occurrence during one method execution, the alias guarantees that \( α \) at program site \( m \) definitely accesses the same memory location as \( β \) at the method entry.

3.3 Side-Effect Computation with Lazy Access Path Resolving

Since the computation of use sets is almost the same as that of the modification sets, this section will only discuss
the computation of modification sets. To perform lazy access path resolving, we should represent the modifications of a method with access paths on formal parameters as much as possible. Formally, the modification set of a method \( m \) can be denoted as \( \langle A_m, L_m \rangle \), in which \( A_m \) is a set of access paths on \( m \)'s formal parameters and \( L_m \) is a set of abstract locations corresponding to the side-effects that can not be represented by set \( A_m \).

For a method with no method invocation, its side-effects come from each definition statement. The first foreach loop in Algorithm 1 performs the side-effect collecting. For each assignment, we firstly check whether the defined access path can be mapped to a must aliased access path at the method entry (on formal parameters). If there does exist an aliased access path, then the aliased one will be added to set \( A_m \) representing the definition targets. Otherwise, the defined access path will be resolved to the actually accessed abstract locations and then added to \( L_m \).

Taking method \( \text{zar}(C) \) in Figure 1(b) for example, \( x.i \) at \( S6 \) and \( y.value \) at \( S7 \) can be mapped to \( c.i \) and \( c.i.value \) at the method entry, respectively. Therefore, the modification set of method \( \text{zar}(C) \) will be \( \{ c.i, c.i.value \} \).

**Algorithm 1**: Side-Effect Analysis

**Input**: \( m : \text{Method} \)

**Output**: \( \langle A_m, L_m \rangle \)

\[
\begin{align*}
\text{let } E_m \text{ be the entry of } m, \text{ and } S &\text{ be a set of all assignments in } m; \\
\text{let } \text{Alias}(m) &\text{ be the interstatement must aliases in } m; \\
\text{let } \text{Loc}(a) &\text{ be the abstract locations accessed by access path } a; \\
\text{let } C &\text{ be a set of all method calls in } m; \\
\text{foreach } "n : \alpha = \ldots" \in S \text{ do} \\
&\text{if } \exists \beta. (E_m; \beta, n; \alpha) \in \text{Alias}(m) \text{ then} \\
&\quad A_m := A_m \cup \{ \beta \}; \\
&\quad \text{else} \\
&\quad L_m := L_m \cup \text{Loc}(a); \\
\end{align*}
\]

\[
\begin{align*}
\text{foreach } c \in C \text{ do} \\
&\text{foreach callee } Q \text{ of } c \text{ do} \\
&\quad \langle A_Q, L_Q \rangle := \text{SideEffect}(Q); \\
&\quad L_m := L_m \cup L_Q; \\
&\quad \text{foreach } \alpha \in A_Q \text{ do} \\
&\quad \gamma := \text{access path on actuals mapped from } \alpha \text{ in } Q; \\
&\quad \text{Treat the access on } \gamma \text{ as an assignment } "c : \gamma = \ldots"; \\
\end{align*}
\]

For method calls, the side-effects of the callees should be merged into the callers. Let \( m \) be the caller, and \( Q \) be the callee. Then the abstract location part of \( Q \)'s side-effects, namely \( L_Q \), could be immediately merged to \( L_m \), the modified abstract location set of method \( m \). While adding \( A_Q \) to method \( m \)'s side-effect sets has two major steps. Firstly, all access paths in \( A_Q \), namely a couple of access paths on \( Q \)'s formal parameters, should be mapped to the access paths on \( Q \)'s actual arguments. Then, to exploit more lazy access path resolving, such access paths in the callsites should also be mapped to the access paths on method \( m \)'s parameters. This step is the same as collecting side-effects from ordinary assignments. The second outmost foreach loop in Algorithm 1 shows the detail of the above operations.

In the example of Figure 1(b), method \( \text{zar}(C) \)'s modification set is \( \{ c.i, c.i.value \} \). Firstly, these two access paths should be mapped to \( \{ b.i, b.i.value \} \) at \( S4 \). Then, \( b.i \) and \( b.i.value \) will be treated as ordinary assignment targets, \( b.i \) will be mapped to access path \( a.i \) at the entry of method \( \text{bar}(C) \) according to the alias \( \langle \text{Entry}_{\text{bar}}; a.i, S4; b.i \rangle \) (\( \text{Entry}_{\text{bar}} \) stands for the entry of \( \text{bar}(C) \)). While \( b.i.value \) can not be mapped to any access path on parameter \( a \). Therefore, it will be resolved to an abstract location \( O_{S3}.value \) and added to the abstract location part of method \( \text{bar}(C) \)'s side-effects (\( O_{S3} \) models all object allocated at \( S3 \)).

In the algorithm, we use interstatement must alias to map an internal access path to an access path on the method interface. Thereby, the side-effects can be propagated in access path form, and the accessed locations of these access paths can be lazily resolved. On the whole, the proposed algorithm has almost the same structure as traditional side-effect analysis except finding and checking the aliases. This makes it quite easy to be adopted into a program analysis system already having some side-effect analysis mechanisms.

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 mode. When there are recursions, we iteratively compute the side-effect sets until a fixed point is reached. By finding the strong connected components in a call graph before analysis, the fixed point searching can be restricted to small areas, and the iterative computation can be minimized.

### 4 Interstatement Must Alias Analysis

Having illustrated the new side-effect analysis approach, now the remaining problem is to find the interstatement must aliases in a program. We design two GVN-based algorithms to do interstatement must alias analysis. One is based on a trivial intraprocedural GVN which only determines the equivalence between local variables, and the other is based on a more elaborate interprocedural GVN. The former has advantages in efficiency, while the later is more precise. Their trade-offs will be investigated in the empirical study.

#### 4.1 Interstatement Must Alias Analysis Based on a Trivial Intraprocedural GVN

For efficiency, this section just uses a trivial global value numbering (GVN) method to do must alias computation. The trivial GVN method works on a SSA form [13]. It can tell whether a local variable in some program site has the same
value as a formal parameter, but makes no attempt to trace
the value of heap accesses, either instance field accesses
or array element accesses. Two occurrences of heap access
\( v_i.f \) can have different values even when variable \( v_i \)
remains unchanged, given that there were an assignment "\( x.f = \ldots \)"
where \( x \) may alias with \( v_i \). Tracing the value of heap accesses
demands more calculation, and in the trivial GVN method,
we just conservatively assume that any heap access could
have different values in different program sites.

Algorithm 2 formalizes the process of interstatement must
alias computation based on the trivial GVN method. In the
algorithm, \( v.\delta \) is an access path starting from variable \( v \)
with one level of field or array dereference. \( v =_{gvn} p \) means that
variable \( v \) shares the same value number as formal parameter
\( p \). For the occurrence of \( v.\delta \) at statement \( n \), according to the
basic properties of GVN and SSA form, it is clear that \( v \) in \( n \)
has the same value as \( p \) in the method entry \( E_m \). When the
program flow is from \( E_m \) to \( n \), access path \( p.\delta \) in the latest
occurrence of \( E_m \) always accesses the same location as \( v.\delta \)
in \( n \). Therefore, \( (E_m; p.\delta, n; v.\delta) \) is a forward must alias.

Algorithm 2: Alias Analysis Based on a Trivial GVN

\[
\text{Input: } m : \text{Method} \\
\text{Output: } \text{Alias}(m) \rightarrow \text{the interstatement must aliases in } m \\
\text{let } E_m = \text{the entry of } m, \text{and } S = \text{a set of all assignments in } m; \\
\text{let } \text{Param}_m = \text{the parameter set of } m; \\
\text{foreach } \langle n : v.\delta = \ldots \rangle \in S \text{ do} \\
\text{foreach } p \in \text{Param}_m \text{ do} \\
\text{if } v =_{gvn} p \text{ then} \\
\quad \text{Alias}(m) = \text{Alias}(m) \cup \{E_m; p.\delta, n; v.\delta\}; \\
\text{end} \end{array}
\]

In method \( \text{zar}(C) \) of Figure 1(b), the equivalence
between \( x \) and \( c \) can be determined by the trivial GVN
method. Therefore, we can obtain a forward must alias
\( \langle \text{Entry}_{zar}, c.i, S_6: x.i \rangle \) and map the definition on \( x.i \) to a
definition on \( c.i \). However, a trivial GVN has difficulties in
finding the equivalence between \( y \) and \( c.i \). It cannot identify
the forward must alias \( \langle \text{Entry}_{zar}, c.i, value, ST: y, value \rangle \),
and thus cannot map the definition on \( y, value \) to the defi-
nition on \( c.i, value \). In the next section, we will propose a
more elaborate GVN method. It is designed for must alias
computation and can identify all the above aliases.

4.2 Interstatement Must Alias Analysis Based on
a More Elaborate Interprocedural GVN

Traditional GVN techniques mainly work for a single
procedure, which may lead to precision loss in interpro-
dural alias analysis. To provide better support for the must
alias computation, we present a new interprocedural GVN
technique based on the famous hash-based GVN method [7].

4.2.1 The Extended SSA Form

The hash-based GVN is also founded on the SSA form.
A typical SSA form only models assignments to local vari-
ables. To handle heap accesses more precisely, we adopt the
SSA extension from Fink et al.’s work [8]. In the extension,
an instance field access \( l.f \) is transformed to array access
\( f[l] \), while a Java array access \( a[i] \) is transformed to \( H[a][i] \),
where \( H \) stands for the memory space accessed by array \( a \).
All arrays possibly accessing the same memory space share
the same virtual array \( H \). In the extended SSA form, two
expressions like \( X[i_1][i_2] \ldots [i_n] \) at a program site definitely
have the same value if the array spaces are the same and all
the indexes are equal. Figure 3 illustrates a simple example
of the extended SSA form. In the example, array \( a_1 \) and \( a_2 \)
are two versions of \( a \) resulting from different assignments.
\( H \) represents both the memory space accessed by \( a_1 \) and
the memory space accessed by \( a_2 \). \( H_1 \) and \( H_2 \) denote two
possible states of the array space \( H \). The array \( f \) in \( S5 \) and
\( S6 \) can be regarded as a static array. It represents a memory
space separate from \( H \).

\[
\begin{array}{c}
a = \text{new int}[10]; \\
a_1 = \text{new int}[10]; \\
S1: a[i] = 0; \\
S1: H_1[a_1][i] = 0; \\
a = \text{new int}[20]; \\
a_2 = \text{new int}[20]; \\
S2: j = a[i]; \\
S2: j = H_1[a_2][i]; \\
S3: a[i] = 2; \\
S3: H_2[a_2][i] = 2; \\
S4: k = a[i]; \\
S4: k = H_2[a_2][i]; \\
r = \text{new C}; \\
r = \text{new C}; \\
S5: r.f = 1; \\
S5: f_1[r] = 1; \\
S6: r.f = 2; \\
S6: f_2[r] = 2; \\
\end{array}
\]

Figure 3. An Example for the Extended SSA Form

4.2.2 Basic Model of the GVN Method

The new GVN method can be formalized into a tuple
\( (E, \Gamma, N, P) \). Let \( S \) be a set of all statements in the SSA
form and \( Expr \) be a set of all expressions. \( E: S \rightarrow 2^{Expr} \)
is a map from a statement to the expressions needing to
be numbered at that site. \( E(s) \) firstly contains all access
paths whose alias relations are cared at statement \( s \) (often
just the occurred ones). Since the access paths can be array
accesses, \( E(s) \) also contains all expressions possibly acting
as array indexes. Besides, in order to perform interprocedural
GVN, for a call statement, \( E(s) \) contains all actual access
paths (i.e., access paths on actual arguments) which are
possibly read or written by the call. For a method entry or
exit, \( E(s) \) contains all formal access paths (i.e., access paths
on formal parameters) possibly accessed inside the method,
so that the relations between the access paths in the method
body and the access paths in the method entry (exit) can
be inferred. Due to the possible recursion in data structures,
here all access paths on a variable \( v \) only mean all access
paths starting from \( v \) and with length up bounded to \( k \). This
limitation makes sure the analysis terminates.

In \( (E, \Gamma, N, P) \), \( \Gamma: S \rightarrow 2^{Expr \times N} \) is a map from state-
m ents to the numbering states of all expressions (number-
ing context). The numbering context at a statement \( s \) is \( \Gamma(s) \). \( \Gamma(s)(e) \) stands for the value number of expression \( e \) at statement \( s \). To precisely record the whole value numbering phase, we also use \( \Gamma_-(s) \) and \( \Gamma_+(s) \) to denote the numbering states before \( s \) and after \( s \), respectively.

Numbering function \( N \) calculates a value number for a given expression under a given numbering context. This is almost the same as traditional hash-based GVN. Function \( \mathcal{P} : S \rightarrow N \) gets the sequence number of a given statement. For statements \( a \) and \( b \), \( \mathcal{P}(a) < \mathcal{P}(b) \) indicates \( a \) is processed before \( b \). Since the numbering of expressions in the later statements depends on the numbering of expressions in the former statements, the sequence number can be used to trace value changes, and thereby facilitate alias computation.

### 4.2.3 Interprocedural Value Numbering

For method calls, a major problem in interprocedural value numbering is determining the number of each expression after the call. Traditional GVNs ignore the equivalence relations introduced by a call and therefore may lose precision. Actually, the GVN results of a callee can reflect its effects in some way. Adopting the callees’ numbering results into the caller’s value numbering phase and performing interprocedural analysis may achieve better results.

To facilitate interprocedural value numbering, this paper introduces the GVN summary to represent the equivalence relations produced by a method. For a method \( M \), let \( \mathcal{I}_M \) be a collection of interface access paths starting from \( M \)’s formal parameters or return value. The GVN summary is a map \( \xi : \mathcal{I}_M \rightarrow \text{Expr} \), where its range is the access paths on formal parameters together with a special symbol \( \perp \) denoting uncertain access paths. The map from an access path \( \alpha \) to an expression \( e \) denotes that after method execution \( \alpha \) has the same value as \( e \) at the method entry whatever path is passed. Figure 4 illustrates an example of the GVN summary. In this summary, \( a.f \mapsto b.f \) indicates after method execution \( a.f \) definitely has the same value as \( b.f \) at the method entry. \( c.f \mapsto \perp \) indicates the value of \( c.f \) is uncertain after the execution of method \( \text{set()} \). \( \text{return} \mapsto b \) indicates the return value is always parameter \( b \).

```plaintext
T set(T a, T b,T c){
   a.f = b.f;    // a.f \mapsto b.f
   c.f = null;   // c.f \mapsto \perp
   return b;    // \text{return} \mapsto b
}
```

**Figure 4. Method set() and its GVN Summary**

Let \( \text{actual}(e) \) be the actual access path corresponding to a formal access path \( e \). Given a method call, if it has only one target, then for the expressions not affected by the call, their value numbers remain unchanged. While for those may be affected by the call, their value numbers should firstly be determined by the following two rules:

\[-\alpha \mapsto e_0 \Rightarrow N(\text{actual}(\alpha)) = N(\text{actual}(e_0))\]

\[-\alpha \mapsto \perp \Rightarrow N(\text{actual}(\alpha)) = \text{new number}\.\]

The former rule claims that if access path \( \alpha \) has the same value as formal access path \( e_0 \) at the method entry, then after method call, actual access path \( \text{actual}(\alpha) \) will have the same value as \( \text{actual}(e_0) \). The later rule indicates if access path \( \alpha \) has uncertain value, then after the call, the actual access path corresponding to \( \alpha \) will be assigned with a new generated value number. For those who can not get their value numbers from the above two rules, they will also be assigned with new numbers.

For a virtual call with more than one possible target, we firstly map the post-call expression \( e \) to a pre-call expression \( e' \) according the equivalence relations produced by each callee. After this step, if under all callees, expression \( e \) can be mapped to the same \( e' \), then \( e \) will inherit the value number of \( e' \); otherwise we also introduce a new number for \( e \) to show that the value of \( e \) is unclear.

The GVN summary of a method can be collected in a straight-forward way. We just need to check each access path in \( \mathcal{I}_M \) at the method exit and find the formal access path equal to it at the method entry. All the methods are analyzed in a bottom-up order. Each method will be analyzed only once. During the analysis of a method, if one of its callee \( Q \) is not yet analyzed, we let its GVN summary to be:

\[\{e \mapsto \perp | e \in \mathcal{I}_Q, \text{ and } e \text{ may be affected by method } Q\}\]

This is a conservative approximation for \( Q \), which will make every possibly affected expression get a new value number.

Inside a method, the GVN process is carried out in a way similar to Click’s approach [7]. All statements are processed in reverse post order, and all relevant expressions are numbered. For a used expression \( e \), when calculating its value number, we firstly check whether there is another expression \( \tilde{e} \) resulting in the same value as \( e \) from the hash-table. If so, \( e \) will inherit \( \tilde{e} \)’s value number, otherwise \( e \) will get a new generated value number. For a modified expression, its value number is determined by the assignment or the callees’ GVN summaries.

### 4.2.4 Identifying the Interstatement Must Aliases

We can immediately get a couple of interstatement must aliases according to the GVN results. Theorem 1 presents an important sufficient condition for identifying must aliases.

**Theorem 1.** Let \( a, b \) be two program sites and \( X_a, X_b \) be two SSA numbered versions of array space \( X \). If \( P(a) < P(b) \) and \( \forall i \in \{1, \ldots, n\}, \Gamma_+(a)(\alpha_i) = \Gamma_-(b)(\beta_i) \), then \( \langle a: X_a[\alpha_1] | \alpha_2 | \ldots | \alpha_n, b: X_b[\beta_1] | \beta_2 | \ldots | \beta_n \rangle \) is a forward must alias.

**Proof.** Firstly, we can prove that for program sites \( a \) and \( b \), if \( P(a) < P(b) \) and \( \Gamma_+(a)(x) = \Gamma_-(b)(y) \), then variable \( x \) in the latest occurrence of \( a \) has the same value as variable \( y \) in the current occurrence of \( b \). According to this observation and the provided conditions, in the latest occurrence of \( a \)
before b, the array indexes of expression \(X_a[\alpha_1] \ldots [\alpha_n]\) definitely have the same values as the corresponding indexes of expression \(X_b[\beta_1] \ldots [\beta_n]\) in current point b. Since \(X_a\) and \(X_b\) share the same array space, these two expressions definitely access the same memory location. According to the definitions, \(\langle a: X_a[\alpha_1][\alpha_2] \ldots [\alpha_n], b: X_b[\beta_1][\beta_2] \ldots [\beta_n]\rangle\) is a forward must alias. Therefore, Theorem 1 can be proved. \(\square\)

```java
void zar(C c) {
    C x0 = c;
    x0 \rightarrow 1
    S5 Integer y0 = i0[x0];
    i0[x0] \rightarrow 2, y0 \rightarrow 2
    S6 i1[x0] = null;
    i1[x0] \rightarrow 4
    S7 value1[y0] = 1;
    value1[y0] \rightarrow 5
}
```

**Figure 5. The GVN Results**

For method `zar(C)` in Figure 1(b), its GVN results are depicted in Figure 5, where i0, i1 are two versions of array space i and value0, value1 are two versions of array space value. According to the numbering results and Theorem 1, we can immediately find forward must aliases:  
\(\langle Entry_{zar}: i0[c], S5: i0[x] \rangle, \langle Entry_{zar}: i0[c], S6: i1[x0] \rangle, \text{ and } \langle Entry_{zar}: value0[i0[c]], S7: value1[y0] \rangle\). Transforming the array accesses back into instance field accesses, we will get forward must aliases \(\langle Entry_{zar}: c.i, S5: x0.i \rangle, \langle Entry_{zar}: c.i, S6: x0.i \rangle, \text{ and } \langle Entry_{zar}: c.i.value, S7: y0.value \rangle\), with which the definitions on \(x0.i\) and \(y0.value\) can be mapped to definitions on \(c.i\) and \(c.i.value\).

5 Empirical Study

To validate the proposed side-effect analysis method, we implement it on Soot 2.3.0 [21] with SPARK [11] inclusion-based context-insensitive points-to-analysis as the base. Soot already has a simple global value numbering tool. Therefore, we only need to implement a new side-effect analysis algorithm and a new interprocedural GVN algorithm. The following experiments are based on Soot’s simple GVN and the new GVN designed for alias computation.

5.1 Experiment Settings

Table 2 presents the benchmark programs used in our empirical study. The benchmark suite firstly includes soot-c from the Ashes suite [1], polyglot 1.3.5 [14] used by Soot 2.3.0, and 6 benchmarks from the DaCapo benchmark suite version 2006-10-MR2-xdeps and version beta-2006-08 (only ps from this version) [4]. Besides, we also use 213x_javac and 202_jess from the free available SPEC jvm2008 benchmark [2] (They are two programs used as the input of program compress). All these programs, or their previous versions, have been used as benchmarks in other researches on pointer analysis or side-effect analysis.

The analysis is performed on Soot’s Simple intermediate representation with Java 1.4.1 as the linked library.

As a prototype, we have not considered the side-effects of native methods, but such methods do be simulated during SPARK pointer analysis. In Table 2, Column Classes counts the classes analyzed by Soot. Column R-Methods counts the reachable methods in SPARK’s call graph, and Column SparkTm shows the time consumed by SPARK points-to analysis. Three algorithms are studied on the experiment settings. In the algorithms, OLD represents the traditional location-based side-effect analysis. TGVN and IGVN represent our new side-effect analysis based on the trivial GVN and the newly developed interprocedural GVN, respectively. In IGVN, we set the up-bound of access path length to 1, namely only a single field or array access is allowed on a reference variable during the value numbering phase.

All the analysis data are collected on a PC with 2.8GHz Pentium CPU and 2G RAM. Besides the analysis time, 4 metrics are used to assess the effect of each algorithm. One is the average size of each method’s side-effect sets, which is the average size of modification sets plus the average size of use sets. Note that for comparison, in the final side-effect sets, all access paths have been resolved to the accessed locations. Each field of an abstract object is counted as one location, and the global locations are not counted. The other 3 metrics are the methods with part of their read/write effects represented as access paths (Column P-Clear), the methods with all their read/write effects represented as access paths (Column Clear), and the methods with side-effect analysis precision improved by more than 50%.

5.2 Experimental Results

Table 3 presents the experimental results. From the table, we can see that for each benchmark, the new method averagely reduces about 5.5% of a method’s side-effect sets, and 12.5% of methods have their side-effect analysis precision improved by more than a half. 78% of methods have their read/write effects partly represented as access paths, in which 22% of methods have all their read/write effects represented as access paths.

Even though there do exist some improvements, the results in Table 3 are not encouraging. We investigate the reason, and find 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. To suffer less from them, we make a slight relaxation on the soundness requirement and adopt two heuristics to improve the analysis. The first is that we ignore the implicit class initialization calls and finalize calls. These calls are often occasional and not kept in mind during method implementation. Ignoring them usually does not affect the understanding of a method’s behavior. The other is that we consider `java.lang.Object`, `java.lang.String`, `java.lang.Class`, and the wrapper classes corresponding to `char`, `int`, `float`,...
The experiments also show that a simple intraprocedural
and so on as build-in types. Such classes are immutable types,
and objects of immutable types are values instead of variable
memory locations. It is reasonable to treat them in the
same way as build-in types, and an access to a build-in type
value is generally not considered as the read/write effects of
a method. These heuristics are easily acceptable. They make
the analysis less safe, but may still benefit many applications
like program understanding and software maintenance.

Table 4 presents the results of the experiments using the
above heuristics, in which Column Reach lists the new reach-
able method counts without considering class initializations,
finalize calls and methods of immutable types. From this ta-
ble, we can see that now the new method averagely reduces
over 16.7% of each method’s read/write effects, and over
26% of methods have their side-effect sets reduced by more
than a half. The efficiency of the new approach also signifi-
cantly improves. Although it still averagely demands 7 times
of the old analysis time, this time is more endurable since the
precision. However, for methods in small call stack depth
(e.g., entry method main()), the side-effects of their callees
under different calling contexts tend to be merged. Therefore,
such method will get less improvement in the new side-
effect analysis. Unfortunately, the side-effect sets of these
methods are often huge. This prevents us from making more
refinement on all method’s average side-effects.

Table 3 and Table 4 show the experimental results on
both modification sets and use sets. Besides, we have to
notice that generally there is more precision improvement
in modification set computation than that of the use sets
(see Table 5). To be more efficient, one can only apply the
proposed technique to write effect analysis, while the read
effects can be analyzed in traditional location-based fashion.

Even though 99.9% of methods have their side-effect sets
reduced, there do exist some methods (around 5 methods
in each benchmark program) whose modification/use sets
increase. We investigate these methods and find that the root
cause is the type masks in SPARK pointer analysis. Since
we map a memory access inside a method to the access via
other required analyses). The experimental results suggest that the
proposed approach might be more valuable in a task where
safety is not always critical.

In Table 4, the percentage of clear methods also increases.
A method tagged clear in Table 3 and Table 4 means all its
reads and writes can be represented as access paths. However,
the clearness of a method does not always mean it really has
read or write effects, because an access path may access no
location given that some points-to sets are empty. The clear-
ness property is especially useful in program understanding.
It not only states the behavior of a method, but also can avoid
the situation that marks a setter method to be side-effect free
only because of some empty points-to sets.

5.3 Discussions

Actually, our new method adds some context-sensitivity
to the side-effect computation and thereby can achieve more
precision. However, for methods in small call stack depth
(e.g., entry method main()), the side-effects of their callees
under different calling contexts tend to be merged. Therefore,
such method will get less improvement in the new side-
effect analysis. Unfortunately, the side-effect sets of these
methods are often huge. This prevents us from making more
refinement on all method’s average side-effects.

Table 3. The Results of the Side-Effect Analysis

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Classes</th>
<th>R-Methods</th>
<th>SparkTime(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>soo-t</td>
<td>An analysis framework for Java</td>
<td>2057</td>
<td>7424</td>
<td>52.3</td>
</tr>
<tr>
<td>polyglot</td>
<td>A framework for Java language extensions</td>
<td>1626</td>
<td>5284</td>
<td>53.7</td>
</tr>
<tr>
<td>antlr</td>
<td>A parser and lexical analyzer generator</td>
<td>1516</td>
<td>5874</td>
<td>44.8</td>
</tr>
<tr>
<td>jython</td>
<td>A Python interpreter</td>
<td>1954</td>
<td>8748</td>
<td>53.9</td>
</tr>
<tr>
<td>bloat</td>
<td>A Java bytecode optimizer</td>
<td>1684</td>
<td>7833</td>
<td>56.8</td>
</tr>
<tr>
<td>pmld</td>
<td>A Java source code analyzer</td>
<td>2167</td>
<td>7119</td>
<td>48.5</td>
</tr>
<tr>
<td>ps</td>
<td>A postscript interpreter</td>
<td>3377</td>
<td>8170</td>
<td>66.5</td>
</tr>
<tr>
<td>fop</td>
<td>An output-independant print formatter</td>
<td>4942</td>
<td>7719</td>
<td>73.6</td>
</tr>
<tr>
<td>213x_javac</td>
<td>JDK Java compiler</td>
<td>1553</td>
<td>6257</td>
<td>58.8</td>
</tr>
<tr>
<td>202_jess</td>
<td>A Java expert shell system</td>
<td>3345</td>
<td>7381</td>
<td>58.9</td>
</tr>
</tbody>
</table>

Table 2. The Benchmark Programs

<table>
<thead>
<tr>
<th>Name</th>
<th>Time(s)</th>
<th>Average Side-Effects</th>
<th>P-Clear</th>
<th>Clear</th>
<th>+50%</th>
</tr>
</thead>
<tbody>
<tr>
<td>soo-t</td>
<td>25.0</td>
<td>159.1</td>
<td>250.3</td>
<td>10952</td>
<td>10190</td>
</tr>
<tr>
<td>polyglot</td>
<td>16.6</td>
<td>115.4</td>
<td>186.3</td>
<td>10065</td>
<td>9739</td>
</tr>
<tr>
<td>antlr</td>
<td>18.1</td>
<td>125.7</td>
<td>191.1</td>
<td>11458</td>
<td>10697</td>
</tr>
<tr>
<td>jython</td>
<td>24.8</td>
<td>489.5</td>
<td>715.8</td>
<td>16907</td>
<td>16534</td>
</tr>
<tr>
<td>bloat</td>
<td>35.5</td>
<td>472.0</td>
<td>692.3</td>
<td>16205</td>
<td>14726</td>
</tr>
<tr>
<td>pmld</td>
<td>21.6</td>
<td>161.2</td>
<td>265.5</td>
<td>12519</td>
<td>11882</td>
</tr>
<tr>
<td>ps</td>
<td>32.7</td>
<td>383.5</td>
<td>731.2</td>
<td>12915</td>
<td>12521</td>
</tr>
<tr>
<td>fop</td>
<td>24.4</td>
<td>443.3</td>
<td>778.1</td>
<td>13521</td>
<td>12787</td>
</tr>
<tr>
<td>213x_javac</td>
<td>16.7</td>
<td>134.2</td>
<td>229.3</td>
<td>12757</td>
<td>11903</td>
</tr>
<tr>
<td>202_jess</td>
<td>22.4</td>
<td>298.8</td>
<td>565.9</td>
<td>12876</td>
<td>12151</td>
</tr>
<tr>
<td>Average</td>
<td>×11.2</td>
<td>×18.6</td>
<td>+5.5%</td>
<td>+5.8%</td>
<td>78.7%</td>
</tr>
</tbody>
</table>

**Table 3. The Results of the Side-Effect Analysis**
### Related Work

There is a large body of work on side-effect analysis for languages with pointers. Ryder et al. [18] present a general framework for modification side-effect analysis of C programs. Clausen [6], Razafimahefa [16], Rountev [17], Milanova et al. [12], Le et al. [10], and Xue et al. [22] study the side-effect analysis problem of Java programs based on various pointer analysis algorithms (or type-based analysis algorithms). Their work leads to many interesting findings. However, all those approaches compute side-effects in a traditional location-based fashion. This is obviously different from the access path based approach proposed in this paper. Besides, rather than concentrating on the pointer analysis, we pay more attention to the side-effect analysis itself. Our work just makes better use of pointer information instead of redesigning any new pointer analysis algorithm.

Sălcianu and Rinard [20] also compute side-effects in the location-based fashion. They do not directly represent side-effects as access paths. But since their object representation is abstracted according to the access paths, the approach also could be viewed as an access path based one. The major differences between our work and theirs are that, first, we directly represent side-effects as access paths, and second, our approach works on an inclusion-based context-insensitive pointer analysis while theirs works on a special summary based one. Their pointer analysis is context-sensitive and hence might be more precise, but there are still many systems preferring simple IBCI pointer analysis and these are the places where our approach may be applied. Our objective is refining the side-effect analysis under IBCI pointer analysis without losing much scalability, and the experimental results show that this objective has at least been achieved to some extent. Sălcianu and Rinard’s approach also can represent the write effects of a method as regular expressions. This is somewhat similar to representing side-effects as access paths. However, they just use regular expressions to show why a method is impure instead of incorporating them to refine side-effect computation. Their regular expressions are mainly inferred from a points-to graph, while our side-effect representation is inferred by GVN. In IBCI pointer analysis, the points-to graphs are often large. Inferring regular expressions starting from formal parameters for points-to graph nodes demands more effort than finding access path based side-effect representation with GVN, and mapping a regular expression with quantifiers like "*" back into abstract locations is also non trivial. Therefore, although powerful, the regular expressions have many difficulties to be adopted into our style of side-effect collecting.

Recently, Cherem and Rugina [5] propose a method to build lightweight method summaries by escape and effect analysis. The approach also represents the read/write effects with a form similar to access paths. However, it can not precisely represent read/write effects on objects loaded from static fields and objects loaded from a parameter with access-depth beyond a given limit (all such objects are modeled by a single top value ⊤). Our approach does not suffer from the same problem as the side-effects can at least be represented with abstract locations, even though we prefer to represent them with access paths. Cherem and Rugina use a unification-based context-sensitive approach to collect side-effects, while our approach is based on an inclusion-based context-insensitive pointer analysis. It is not clear which style

---

**Table 4. The Results of the Side-Effect Analysis with Heuristics**

<table>
<thead>
<tr>
<th>Name</th>
<th>Reach</th>
<th>Time(s)</th>
<th>Average Side-Effects</th>
<th>P-Clear</th>
<th>Clear</th>
<th>+50%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>O₁D</td>
<td>Tvnn</td>
<td>kVnn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>soot-c</td>
<td>6272</td>
<td>13.4</td>
<td>69.0</td>
<td>168.2</td>
<td>5651</td>
<td>4521</td>
</tr>
<tr>
<td>polyglot</td>
<td>4124</td>
<td>4.6</td>
<td>23.7</td>
<td>41.5</td>
<td>4858</td>
<td>4072</td>
</tr>
<tr>
<td>antlr</td>
<td>4705</td>
<td>10.0</td>
<td>55.2</td>
<td>51.2</td>
<td>5967</td>
<td>4956</td>
</tr>
<tr>
<td>jython</td>
<td>7671</td>
<td>17.6</td>
<td>185.9</td>
<td>382.4</td>
<td>9982</td>
<td>9039</td>
</tr>
<tr>
<td>bloat</td>
<td>6685</td>
<td>18.2</td>
<td>124.4</td>
<td>279.9</td>
<td>10202</td>
<td>8502</td>
</tr>
<tr>
<td>pmd</td>
<td>5953</td>
<td>10.3</td>
<td>46.6</td>
<td>83.7</td>
<td>6181</td>
<td>5048</td>
</tr>
<tr>
<td>ps</td>
<td>6830</td>
<td>16.3</td>
<td>192.9</td>
<td>335.9</td>
<td>5691</td>
<td>4694</td>
</tr>
<tr>
<td>fop</td>
<td>6327</td>
<td>13.8</td>
<td>117.4</td>
<td>286.4</td>
<td>6071</td>
<td>4984</td>
</tr>
<tr>
<td>213x_javac</td>
<td>5119</td>
<td>8.5</td>
<td>49.1</td>
<td>98.8</td>
<td>6746</td>
<td>5583</td>
</tr>
<tr>
<td>202_jess</td>
<td>6058</td>
<td>12.3</td>
<td>104.4</td>
<td>214.7</td>
<td>6029</td>
<td>5030</td>
</tr>
</tbody>
</table>

**Table 5. Average Modification Sets**

<table>
<thead>
<tr>
<th>Name</th>
<th>Average MOD</th>
<th>Average MOD+heuristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>soot-c</td>
<td>5554</td>
<td>5050</td>
</tr>
<tr>
<td>polyglot</td>
<td>5369</td>
<td>5167</td>
</tr>
<tr>
<td>antlr</td>
<td>5869</td>
<td>5332</td>
</tr>
<tr>
<td>jython</td>
<td>8834</td>
<td>8558</td>
</tr>
<tr>
<td>bloat</td>
<td>8567</td>
<td>7401</td>
</tr>
<tr>
<td>pmd</td>
<td>6326</td>
<td>5919</td>
</tr>
<tr>
<td>fop</td>
<td>6773</td>
<td>6261</td>
</tr>
<tr>
<td>213x_javac</td>
<td>6718</td>
<td>6031</td>
</tr>
<tr>
<td>202_jess</td>
<td>6792</td>
<td>6274</td>
</tr>
</tbody>
</table>

\(\times 1.0, \times 1.9\) +16.7%, +16.9%, +82.4%, +83.7%, +26.5%, +29.3%, +26.2%, +27.0%
of analysis is better in scalability and precision. But since our objective is to improve the side-effect analysis building on an IBCI pointer analysis like Spark in Soot, which is scalable, easy for implementation, and widely used, instead of presenting a new building-from-scratch side-effect analysis approach outperforming other context-sensitive ones, it is clear that we have made our own contribution.

Escape information (e.g., [5, 20]) can be used to filter out method-local objects during side-effect propagation and thereby improve the side-effect analysis precision. However, refining side-effect analysis with escape information is on a different direction from our work, and these two approaches are generally not comparable. Even though, it is clear that our method can filter out superfluous side-effects caused by escapable objects. Besides, the method proposed in this paper is simpler. Its does not demand on a new pointer analysis implementation, and the GVN tool can be provided by many program analysis infrastructures.

7 Conclusion

This paper presents a lightweight approach exploiting lazy access path resolving to improve the side-effect analysis precision under inclusion-based context-insensitive pointer analysis. The basic idea of the approach is that a method’s side-effects can be partly represented as access paths on formal parameters. Since in inclusion-based pointer analysis, points-to sets in the callers tend to be smaller than the ones in the callees, by lazily resolving the accessed locations of access paths when propagating a callee’s side-effects to the callers, more precision can be achieved.

We use interstatement must aliases to safely map the reads and writes inside a method to access paths on the method’s interface. Two GVN-based algorithms are designed for alias analysis, one based on a trivial GVN without tracing the value of heap accesses and the other based on a more elaborate interprocedural GVN designed for alias computation.

The empirical studies show that the proposed approach is effective in improving the precision of side-effect analysis. The improvement is particularly significant when the soundness requirement can be slightly relaxed and two reasonable heuristics are used. This suggests the approach might be more valuable in a task like software maintenance where safety is not always critical. The experiments also show that although an elaborately designed GVN can lead to more precision, a trivial GVN would be more practical for the new side-effect analysis.

Acknowledgments

This work is supported by the National High Technology Research and Development of China (863 Program) (2009AA01Z147), NSFC (90818027, 60873049), and China National Defense Key Laboratory of Integrated Avionics System, and China Aviation Science Foundation (No. 20085552021).

References