View metadata, citation and similar papers at core.ac.uk

brought to you by CO

provided by KIT

Universität Karlsruhe - Fakultät für Informatik - Bibliothek - Postfach 6980 - 76128 Karlsruhe

Improving and Evaluating the Scalability of Precise System Dependence Graphs for Object-oriented Languages

Jürgen Graf

Interner Bericht 2009-14

Universität Karlsruhe (TH)
Forschungsuniversität · gegründet 1825

ISSN 1432-7864

Fakultät für **Informatik**

Improving and Evaluating the Scalability of Precise System Dependence Graphs for Object-oriented Languages

Jürgen Graf (graf@kit.edu)

Karlsruhe Institute of Technology, Germany

Abstract. Static program analysis helps to answer questions about program properties such as if a given program does leak confidential information. Besides the well-known approaches using type-systems another way using dependence graphs seems to be promising. Once they are created they improve analysis precision significantly, but their creation suffers from poor performance. We evaluated the precision and scalability of dependence graph creation for object oriented languages and developed a new way to model interprocedural dependencies. This model improves time and memory consumption in some cases up to 90% while it maintains the precision of the previous model.

1 Introduction and Motivation

System dependence graphs (SDG) have been developed over the last 20 years and became a standard device to capture structures and dependencies of a program. They are the basis for multiple applications in program analysis, such as slicing [25,13], debugging [9,24], testing [1], complexity measurement [20], model-checking [6] and information flow control [5]. As their precision is crucial to the overall precision of those applications, many extensions have been proposed to improve precision of the SDG for object oriented languages [12,16,30,13,23,17,26,3,4]. Despite these efforts, the computation of precise dependence graphs is still challenging for programs exceeding a certain size, because scalability has not been their main focus.

The creation of system dependence graphs basically can be divided into 2 main parts. An intraprocedural part that covers method local dependencies and structures and an interprocedural part that models the effects on the global state of the program. The computation of the intraprocedural part is quite straight forward and has no scalability issues. Therefore we are going to focus on the phases of the interprocedural part of the computation.

The interprocedural part itself divides into 3 successive phases where each phase needs the result of the preceding phase as input for its own computations. The first interprocedural phase determines for each method which side-effects may be introduced by dependencies through the globally shared heap data. When the locations on the heap have been identified the next phase is going to compute the interprocedural data flow between them. The last phase computes summaries

for each method invocation showing precisely how the input of the method is going to influence the outcome of the execution and eliminates the need to look inside the method. As all phases need the results of their predecessors as input for their own computations it is quite natural that the first phase is crucial because its results influence the subsequent data flow analysis and summary computation.

In this work we evaluated the scalability of state-of-the-art precise SDG creation [4] and finally determined the model for interprocedural parameter passing that is used during the side-effect computation as the main source for huge space and time consumption. Its computation tends to take longer and consume more space the less precise the initial information about the parameters are. But in order to get precise initial information we need to apply a costly analysis of the aliasing situation in first place. This leads finally to a deadlock were either the initial information is too imprecise for the parameter model to be computed in reasonable time or the computation of the initial information itself is taking too long. As a consequence a new parameter model is introduced which removes the scalability deadlock by improving the treatment of imprecise input while maintaining the precision of the previous approach.

Both parameter models are integrated in our SDG creation tool which is based on the program analysis framework WALA [8]. We evaluated dependence graph creation under various options in a benchmark consisting of up to 22 small (100 lines of code) to medium (60000 lines of code) sized programs. Four different points-to analyses provide various levels of precision for the initial parameter information. The results indicate that precise points-to analyses in most cases have a moderate influence on the overall precision [1] (around 4%, but up to 24%) of the dependency graph while they greatly influence the runtime. In addition the chosen abstraction of the parameter passing model has a huge impact on the runtime and memory consumption.

This work consists of two main parts: The description of the new parameter passing model and the evaluation of its influence on scalability and precision of the system dependency graph computation process. Section 2 explains in detail the computation of the two different parameter passing models that were used during evaluation and reasons about their theoretical advantages and disadvantages. The evaluation results of the implementation in Java are presented in section 3. Section 4 concludes with a brief summary and an outlook on future improvements.

2 Using parameter passing to model interprocedural side-effects

Whenever a method is called it potentially alters the global state of the program and may effect computations that are executed afterwards. Side-effect analysis is applied to capture all possible changes to the global state a certain method may

[1] The precision is measured through the average size of a slice in the SDG.

introduce. Side-effects are introduced through changing or reading the state of data stored on the heap, which persistently resides in memory. In the Java language, heap locations can only be reached through field accesses. Manipulating data directly through pointer arithmetics like in the C language is not possible in Java.

In current SDGs, parameter passing is used to model side-effects in a context sensitive way. Points-to and type information is applied to identify additional parameter nodes that precisely model side-effects for each method. These parameter nodes abstract from real heap locations and take into account that heap data may only be reached through subsequent field accesses.

The result of this kind of side-effect analysis consists of two sets of parameter nodes for each method. One set (mod) of parameter nodes represents abstract locations that may have been modified. They will be referred to as additional output parameters of the method. The other set (ref) contains nodes for abstract locations that may have been read. They will be referred to as input parameters.

```
1  class A {
2    int x, y;
3
4    void change(A a) {
5      int i = a.y;
6      i += a.y;
7      a.x = 42 + i;
8    }
9
10
11
12
13   void call(A a) {
14     change(a);
15   }
16 }
```

⇒

```
1  class A {
2    int x, y;
3
4    void change(A a,
5      IN:h1,
6      OUT:h2)
7    {
8      int i = h1;
9      i += h1;
10     h2 = 42 + i;
11   }
12
13   void call(A a) {
14     change(a, IN:a.y, OUT:a.x);
15   }
16 }
```

Fig. 1. Modeling side-effects through parameter passing.

The example in Figure 1 shows how additional parameters are used to model method side-effects. The original Java program is on the left side of the Figure. It contains a class A with two fields x and y as well as a method *change*. This method contains three operations that potentially introduce side-effects. First it takes an object of class A as argument a and reads the content of field y in line 5 and again in line 6. Afterwards field x of the parameter object a is modified in line 7. Both operations access object fields, whose data is visible outside method *change*. Therefore they may introduce side-effects to the program. In order to incorporate this behavior into the method signature additional parameters h_1, h_2

are added. The referenced heap location of field $a.y$ is modeled by a new input parameter h_1 whereas the modified heap location of field $a.x$ is represented by the new output parameter h_2. The extended method is shown on the right side of Figure 1 in lines 4 to 11. By now it only accesses data directly reachable through its parameters and does not contain any more field accesses. Therefore all potential side-effects are captured by the method interface.

At the callsite of *change* in method *call* the new parameters have to be passed into the called method. This leads to the situation that method *call* now has to access fields $a.x$ and $a.y$. Subsequently additional parameter nodes for method *call* will have to be computed by the side-effect analysis. This way the side-effects are propagated interprocedurally in form of additional method parameters across callsites from the callee to the caller.

In the following subsections we present three different methods to compute method side-effects and handle parameter passing. These models differ in the way dependencies between heap located object instances are structured and how interprocedural effects are propagated. At first we start in section 2.1 with a basic parameter model that does not use a stucture between its parameters. In section 2.2 the creation of the parameter model using object trees of Hammer et at. [4] is explained, while in section 2.3 the new graph based model is introduced.

2.1 Heap accessing instructions as parameters

An obvious way to model side-effects with additional parameters is to create an additional parameter for each instruction that may access a value on the heap. This approach is also used by the SDG that comes with the WALA framework [8]. Its main advantages are that its implementation is straight forward and the computation scales quite well in theory. But it does have some drawbacks when it comes to precision and number of the additional parameters. While the additional parameters can be computed very fast, their number can become large and present a struggle for the successing data flow and summary computation phases.

The example in Figure 2 shows how heap accessing instructions may be used directly to create additional parameter nodes. The program on the left side is the same Java program as on the previous example on Figure 1. Method change contains three field accessing operations where two operations read the value of field $a.y$ and one writes a new value to $a.x$. We create a new parameter for each instruction and replace the field accesses with accesses to the corresponding new parameters. The resulting program is displayed on the right side. A notable difference between this result and the one from Figure 1 is that the program has three new parameters instead of only two. Even if the instructions in line 5 and line 6 access the same field, they are two separate instructions and therefore a new parameter is created for both.

A big advantage of this approach is that the instructions them self do not need to be analyzed in detail and new parameters can be created very fast and easy. On the other hand many new parameters may be introduced that effectively have no difference, like the two parameter for the field accesses to the

```
1  class A {                          1  class A {
2    int x, y;                        2    int x, y;
3                                     3
4    void change(A a) {               4    void change(A a,
5      int i = a.y;                   5         IN:getfield@line5,
6      i += a.y;                      6            getfield@line6,
7      a.x = 42 + i;                  7         OUT:setfield@line7)
8    }                                8    {
9                                  ⇒  9      int i = getfield@line5;
10                                    10     i += getfield@line6;
11                                    11     setfield@line7 = 42 + i;
12                                    12   }
13                                    13
14   void call(A a) {                 14   void call(A a) {
15     change(a);                     15     change(a, IN:a.y, a.y,
16   }                                16               OUT:a.x);
17                                    17   }
18 }                                  18 }
```

Fig. 2. Using heap access instructions as additional parameter nodes.

same field in the example. This situation becomes even more evident when those new parameters are propagated interprocedurally.

A natural way to prevent the creation of too many similar parameters is to search for existing similar parameters and only create a new one if none has been found. But when are two parameters similar? The parameter models in the following two subsections provide two different approaches to detect and handle similar nodes. They also include logic to detect side-effects that may not be visible outside the current method and reduce the number of parameter nodes further. Both approaches have in common that they use points-to and type information to support their decisions while the simple parameter model of this section does not need any of that.

2.2 Object trees

In 1998 Liang and Harrold [13] proposed an approach to model the method side-effects in form of object trees where each node of the tree will be used to add an additional parameter to the corresponding method. The main idea behind this approach is that each object o holds a set of field members f. These members can only be accessed when a reference to the object itself is present. Therefore o is called the base object of f. In object trees the base object is modeled as parent of all its field members. As those members themselves may be a reference to another base object a tree structure naturally emerges. In the example in Figure 1 parameter a of method *change* is the base object for the fields $a.x$ and $a.y$.

Each node in an object tree represents a set of possible object instances of a certain type and each child corresponds to a field belonging to the object type of the parent. Note that nodes do not have to be unique. It is possible and also very common that multiple nodes refer to the same set of object instances. Root nodes of an object tree consist of object instances, whose reference is accessible without any indirections. This holds for all references stored on the stack, like the method parameters as well as for static fields. Starting from each root node n a child c_f is added iff one of the objects n is referring to contains a field f and f may be accessed during method execution. The leafs of an object tree correspond to fields of object instances whose data is never read or fields of primitive type, who by definition do not contain any further field references.

The process of adding new children to an object tree is called unfolding. In theory object trees have to be unfolded until only primitive typed fields or unused objects are left. This leads to very large and potentially unlimited size of the object tree, as data structures may be recursive. As a remedy Liang and Harrold set an arbitrary limit to the depth of the object tree. Hammer and Snelting [4] argued that cutting the object tree at a fixed depth is not a safe approximation and proposed a safe unfolding criterion by using points-to information:

"*A child f of node p needs not to be unfolded, iff there exists a node p' in the path from p to the root node where p' and p have identical points-to sets and p' already has a field f*"

This criterion is used for the construction of object trees in the implementation that is going to be evaluated.

For the computation of side-effects all operations that potentially may introduce a side-effect have to be located and analyzed. In Java these are all operations that access a field value and those are quite numerous. Before explaining the algorithm used to create object trees we will first take a look at all operations that have to be considered during side-effect computation and how they can be summarized and categorized.

Field accesses in Bytecode Java Bytecode contains 19 different operations that access field values. Among those operations there are 2 for static fields, 2 for object fields, 1 for the length of an array and 14 to read and manipulate array fields. We simplified this amount of operations and divided heap accesses into only two categories: field references (*field-get*) and field modifications (*field-set*). In Java Bytecode the operations accessing heap locations are static and non-static *field-get* and *-set* operations as well as *array-get* and *-set* operations. They are mapped to abstract *field-get* and *field-set* operations as shown in Table 1. While *field-get* operations only read the value stored in a location on the heap, the *field-set* operations modify the value. Besides this difference both operations have much in common. They share three properties: a points-to set of the base object whose field is accessed, the field itself and the points-to set of the object field (which may be empty if the field stores a primitive type value).

The creation of object graph or object tree nodes is much more simple when using those abstract field access operations, because the number of different cases

Instruction	Instruction Arguments	Abstraction
`getfield`	Base object o, Field f, Type t	field-get$(pts(o),(f,t),pts(o.f))$
`putfield`	Base object o, Field f, Type t, Value v	field-set$(pts(o),(f,t),pts(v))$
`getstatic`	Field f, Type t	field-get$(pts_{static},(f,t),pts(f))$
`putstatic`	Field f, Type t, Value v	field-set$(pts_{static},(f,t),pts(v))$
`?aload`	Array object a, Element type t, Index i	field-get$(pts(a),('[]',t),pts(a[i]))$
`arraylength`	Array object a, Element type t	field-get$(pts(a),('<len>',\textbf{int}),\emptyset)$
`?astore`	Array object a, Element type t, Value v	field-set$(pts(a),('[]',t),pts(v))$

Table 1. We abstract from Bytecode operations [15] and distinguish only between *field-get* and *-set* operations.

is reduced to only two. To achieve this simplification, static field accesses have to be treated like normal object field accesses. Static accesses do not refer to an existing base object, but they are treated as if they all would refer to the same special base object with the points-to set p_{static}. With the additional condition that p_{static} is disjunct to all other points-to sets, static accesses can be treated like normal accesses without being incorrect or losing precision.

Algorithm The algorithm[2] used to compute side-effects using object trees basically divides into 4 steps. The first three steps build a single data structure containing information about all possibly referenced and modified object fields for each method in the program. In the fourth step this data structure is split into two parts. One containing all referenced and the other containing all modified fields. In the parameter passing model referenced fields are interpreted as additional input parameters where global data flows into the current method. Whereas modified fields are interpreted as additional output parameters where modified data leaves the method. The 4 steps are taken in the following order:

1. Build root nodes from method signature.
2. Build nodes for each heap access in the current method.
3. Propagate side-effects from callee to caller methods.
4. Extract separate *mod* and ref sets and create additional method parameters from set elements.

Object tree creation starts with a root set of nodes for each method. These nodes are derived from the signature of the method and consist of nodes for each method parameter as well as the return and possible exception value. As Java uses call-by-value-reference, method parameters can not be used to manipulate the global program state directly. But by accessing their fields they can be used as gates through which side-effects may be propagated to or introduced from the global program state. Let us consider the example program in Figure 3. The program consists of two static methods foo and *bar*. Method *bar* is calling foo using the constant value 23 and a static field s as method parameters. Inside

[2] The appendix in section B.1 contains the pseudocode of the algorithm.


```
1  class A {
2    int i;
3  }
4
5  class B {
6    A a;
7    int x;
8
9    static B s;
10
11   static B foo(int x, B b) {
12     A a = b.a;
13     int i = b.x;
14     a.i = i + x;
15     return b;
16   }
17
18   static B bar() {
19     B b = foo(23, s);
20     return b;
21   }
22 }
```

```
1  static B foo(int, B);
2
3     aload_1
4     getfield a // Field B.a
5     astore_2
6     aload_1
7     getfield x // Field B.x
8     istore_3
9     aload_2
10    iload_3
11    iload_0
12    iadd
13    putfield i // Field A.i
14    aload_1
15    areturn
16
17 static B bar();
18
19    bipush 23
20    getstatic B.s
21    invokestatic B.foo
22    astore_0
23    aload_0
24    areturn
```

Fig. 3. Example program for side-effect computation. The Java source code is on the left side and the byte code instructions resulting from compilation are on the right side.

method foo the fields x and a of the second parameter b are referenced. Using an indirection over the local variable a the field $b.a.i$ is modified in line 14 of the source code. The object tree computation starts by building root nodes for each method according to their signature. A root tree always consists of a single root node $\top$ that is used to structure multiple nodes of the root set as direct successors. All members of the root set can be determined only by looking at the method signature. Therefore the set of root nodes is not going to change during later phases of the object tree computation. Only additional child nodes may be added to them. Figure 4 shows the result of the root tree computation for methods foo and *bar* of the current example. The root set contains all method parameters (x and b in method foo) as well as special nodes for the return value p_{ret}, exception return value p_{exc} and references to static fields p_{static}.

During the second step each heap accessing operation is inspected and a set of object field candidates is emitted. Those candidates are used to add new child nodes to the object tree. A single candidate may lead to multiple additional child nodes as well as there may be no changes to the current object tree, if the structure of the tree already captures the side-effect of the candidate. Each

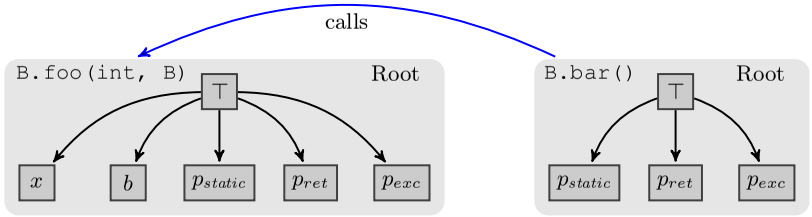

Fig. 4. First step of object tree computation for the example in Figure 3 showing the created root nodes.

candidate holds information about the field that has been accessed and the operation itself. It contains the field name f, the points-to set of the base object *basePts*, the points-to set of the field reference *pts* and the kind of field access *access*. An access may be either modifying *mod* or reading ref the field value.

$$Candidate := (basePts, f, pts, access) :$$
$$Points\text{-}to\ set \times Field \times Points\text{-}to\ set \times \{mod, ref\}$$

The points-to set of the base object *basePts* describes which object instances may have been accessed, while the value of f determines the field that has been accessed. Both informations are combined to identify the object field as precise as possible. The third entry contains the points-to set of the object field itself. It is not used to identify the field, but it may point to an object that is a base object for other field accessing operations. Therefore this information is crucial to recognize transitive dependencies between various field operations where one operation accesses a field that is a reference to the base object of the other operation. The fourth entry of a candidate describes the kind of the field access. As mentioned before we divide field accesses into two basic categories: *field-set* and *field-get* operations. *field-set* operations modify the value of the field they access, so the *access* entry of corresponding candidates is set to *mod*. The *field-get* operations only read the value of the field and so the *access* entry of their candidates is set to ref. It is necessary to remember the kind of field access, as later on either new input or output parameter are created depending on this information.

A candidate is used to add new child nodes to the nodes already present in current object tree. Therefore the object tree algorithm searches all nodes that qualify as potential parent nodes for the candidate. Those are nodes that represent an object that may hold a field reference to the candidate. These nodes p of the current object tree have to be marked as referenced ref, because otherwise the data they contain has never been read and so it may not have been used to access an object field. Additionally their points-to sets have to contain a location that is also referenced by the points-to set of the base object *basePts* of the candidate. Finally the type of the object, a potential parent node refers to, has to contain a field matching the type and name of the candidate field f.

Before the a child node is created at last it is checked for each potential parent node if the unfolding criterion introduced in section 2.2 it met. Only for those who meet the unfolding criterion a new child node is added. The number of those nodes may vary from 0 to the number of all nodes in the current object tree. So a candidate may lead to multiple as well as to no new nodes.

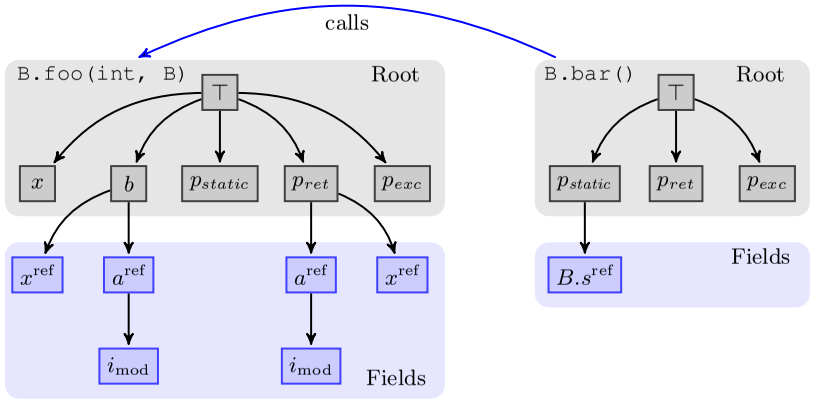

Fig. 5. Object trees for example program in Figure 3 after adding the results of intraprocedural step to the tree of Figure 4.

Consider again methods *foo* and *bar* of the example in Figure 3. They are analyzed intraprocedural by inspecting each instruction for potential field accesses and adding matching child nodes to the current tree. The Java source code may contain multiple instructions per line or even per expression, so we are going to argue on behalf of the corresponding Bytecode, also shown on the right side of Figure 3, to simplify the step-by-step explanation of the object tree structure computation. Starting with method *bar* the first field access operation is a reference to the static field $B.s$ by `getstatic` in line 20. Its source code counterpart is parameter s in line 19. All static field accesses are treated like normal object field accesses to a special base object called p_{static}. So the candidate for the field get operation is $(pts(p_{static}), \text{'}B.s\text{'}, pts(B.s), ref)$. Besides the call to method *foo* no more instructions with potential side-effects are found inside *bar*. So all candidates for new child nodes in method *bar* have been emitted and we move on to method *foo*. Method *foo* contains 3 field accessing operations, two reading a field value `getfield` in line 4 and line 7 of the byte-code and a single write access `putfield` in line 13. The first `getfield` in line 4 reads field a of the parameter b and the second `getfield` operation in line 7 reads a field named x of parameter b. This leads to the corresponding candidates $\{$ $(pts(b), \text{'}a\text{'}, pts(b.a), ref)$, $(pts(b), \text{'}x\text{'}, pts(b.x), ref)$ $\}$. The last field accessing instruction is `putfield`, writing a previously computed value to a field named i.

The base object of the field access is stored in the local variable a and so the resulting candidate is $(pts(a), \text{'i'}, pts(a.i), mod)$. All candidates for the methods foo and bar have been emitted by now and we start searching potential parent nodes for each candidate and create new child nodes for them as they are found. The set of candidates of method bar contains $(pts(p_{static}), \text{'B.s'}, pts(B.s), ref)$ as a single entry. Potential parent nodes are searched in the current object tree of method bar as shown in Figure 4. They have to refer to the same heap location as the points-to set $basePts$ $(= pts(p_{static}))$ of the candidate. This is clearly the case for the root node p_{static}. As p_{static} is the special base object for all accesses to static variables, it may hold a reference to the static field 'B.s'. So a new child node for field 'B.s' is added to p_{static}. Because no other nodes in the current object tree qualify as potential parent nodes of the candidate, the creation of new child nodes for method bar is completed and we continue with method foo. Each of the 3 field accessing operations in method foo emit a corresponding candidate: $(pts(b), \text{'a'}, pts(b.a), ref)$ for the *field-get* in line 4, $(pts(b), \text{'x'}, pts(b.x), ref)$ for the *field-get* in line 7 and finally $(pts(a), \text{'i'}, pts(a.i), mod)$ for the *field-set* in line 13. Starting with the first candidate all nodes in the object tree that may refer to the same heap location as $pts(b)$ are potential parent nodes. Obviously the root node for parameter b qualifies as potential parent. But the root node of the return value p_{ret} qualifies also as potential parent, because foo returns the value of parameter b and thus the return value definitely refers to the same location as b. Both b and p_{ret} refer to an object of class B and as class B has an attribute named a both nodes may have a child node referring to this field. Before a new child is finally added to a parameter node, we have to check if the new child is going to fulfill the unfolding criterion in order to prevent infinite and redundant structures. Children of root nodes always fulfill the unfolding criterion, because there are no nodes in between the path from the child to the root node and thus there can not be an other node that has the same points-to set and refers to the same field as the candidate for the new child node. So a new child node matching the candidate is added to both nodes b and p_{ret}. The second candidate of foo is a read access to the field named x. Again all nodes that may refer to the same heap location as $pts(b)$ are potential parents. As before nodes b and p_{ret} match all criteria and a new child is added to each of them. The last candidate is a modification of the field named i with $pts(a)$ as the points-to set of the base object. There are two nodes in the current object tree that point to the same location as base object of the candidate. These are the newly created field nodes a of the parents b and p_{ret}. Both field nodes represent objects of type A and thus have a reference to a field named i. As the unfolding criterion is also satisfies for both nodes, new child nodes i are added to both. To denote that the field has been written the new nodes receive the subscript mod. Figure 5 shows the object trees of foo and bar after the first iteration of the intraprocedural step.

The previous step computed the intraprocedural side-effects for each method on form of object trees. In the following third step these side-effects are propagated interprocedural. Each method call site is inspected and the side-effects of the callee are propagated to the caller method. For each non-root node n

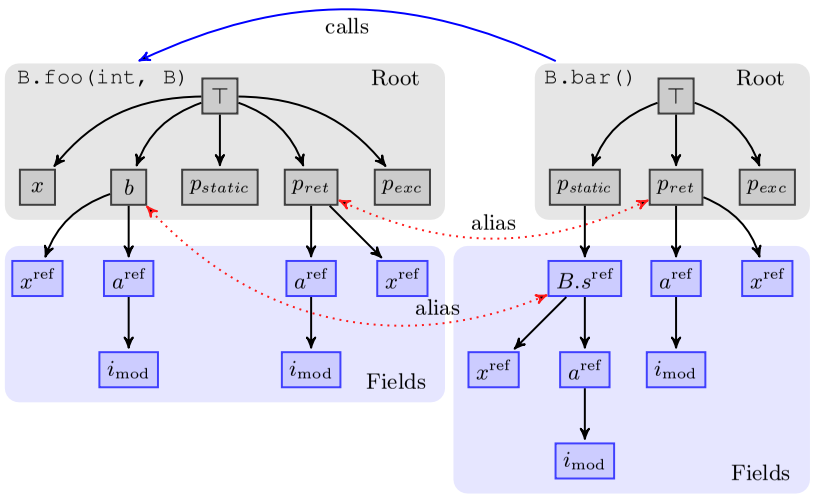

Fig. 6. Object trees for example program in Figure 3 after interprocedural propagation of field nodes.

of the callee object tree a candidate is emitted. The points-to set of the parent of n is used as $basePts$ of the candidate. The field name f, points-to set pts and type of access $access$ are directly taken from n. New nodes are added to the object tree of the caller by searching for matching parent nodes of the candidates as described in the previous step. During the interprocedural propagation for method bar all potential side-effects that may stem from method invocations inside bar are added to the current object tree. Method bar contains a single call site in line 21, calling method foo. So all side-effects that may happen during execution of foo, may also happen during execution of bar. We accomplish this propagation through emitting a candidate for each field node in the object tree of the callee foo. These candidates are used to create new child nodes in the object tree of the caller bar. Figure 5 shows the current state of the object trees for methods foo and bar. The candidates of method foo derived from its field nodes are: $(pts(b), \text{'x'}, pts(x), ref)$, $(pts(b), \text{'a'}, pts(a), ref)$, $(pts(a), \text{'i'}, pts(i), mod)$, $(pts(p_{ret}), \text{'x'}, pts(x), ref)$, $(pts(p_{ret}), \text{'a'}, pts(a), ref)$ and again $(pts(a), \text{'i'}, pts(i), mod)$. The points-to sets of the parameter b in method foo and the static field $B.s$ of method bar are may-aliases because bar calls foo with the static field $B.s$ as parameter b of method foo (Line 19 in Figure 3). So all candidates with the base points-to set of $pts(b)$ are added as new children of node $B.s$ in the object tree of method bar. We add two new children x and a to node $B.s$ and the propagation continues. The next candidate belongs to the modified field named i. The base points-to set of this candidate is the same as the points-to set of the newly created node a, because i is a child of a in the object tree of the callee foo. So a child i for node a is also created in the object

tree of method *bar*. The propagation is not finished by now. The source code in Figure 3 shows that method *bar* stores the return value of method foo into a local variable and uses this variable as its own return value. So the return value of foo is definitely may-aliasing the return value of *bar* and all candidates with the base points-to set of the return value p_{ret} of foo are potential children for the node of the return value p_{ret} of *bar*. The two candidates with $pts(p_{ret})$ as base points-to set match these conditions. New children for field x and field a are added to p_{ret} of method *bar*. Like before a child for the field access to i is added to the newly created node a and the first iteration of the interprocedural propagation is done. The final result is shown in Figure 6.

Step 2 and 3 have to be repeated until a fixed point is reached, because the interprocedural propagation may have added nodes that qualify as parent nodes of intraprocedural candidates. In our example, as the object trees of foo and *bar* changed during the interprocedural propagation phase, the intraprocedural computation has to be run again. But this time nothing has to be added to the object trees and the construction is finished.

The computed object trees contain nodes representing all possible modified and referenced object instances of a method. The final step extracts two separate trees containing only modified (*mod*-tree) or referenced (ref-tree) object instances. It starts with the extraction of the *mod*-tree by including all non-root nodes that have been marked as modified. Subsequently all nodes on a path from a node that is already contained in the *mod*-tree to a root node are included. The extraction is finished when no new nodes can be added to the *mod*-tree.

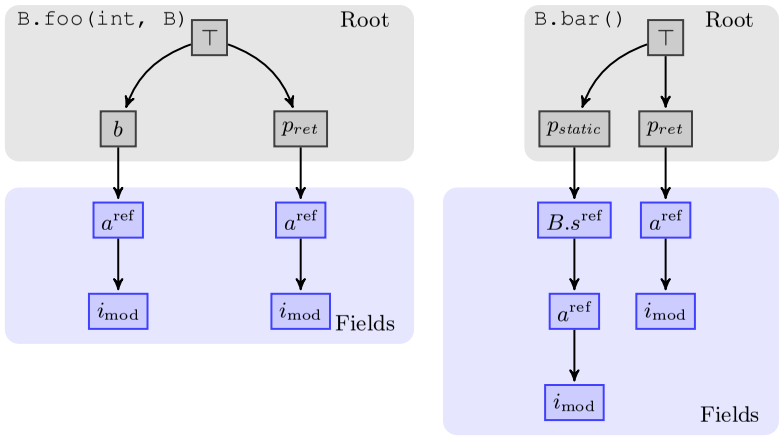

Fig. 7. Object trees for example program in Figure 3 containing only nodes referring to modified object instances.

There are 2 field nodes in each object tree of the current example in Figure 6 that are marked as modified. All four refer to the modified field i. Starting from

these nodes all nodes on the path to the ⊤ root node are included in the *mod*-tree as well, even if they are not marked as modified. The data inside those objects may have not been changed, but the data in one of their fields probably has. So if a member of an object has been changed, the object itself has been changed too. By including these elements in the *mod*-tree we remember the access path through which field accesses the side-effect may have been introduced. So we do not only know what side-effects may happen, but also how they may be introduced. The resulting *mod*-trees of the current example are shown in Figure 7.

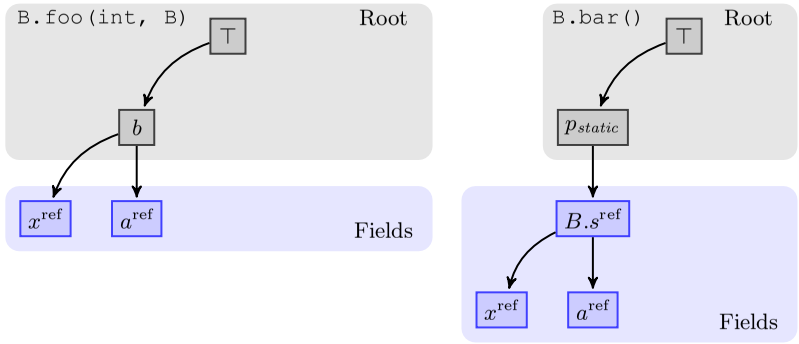

Fig. 8. Object trees for example program in Figure 3 containing only nodes belonging to referenced object instances.

On the other hand the ref-tree contains all nodes referring to data that may have been passed from the outside into the current method. Basically all nodes that could not have been used to pass data into the method in the first place have to be removed beforehand. So all nodes in the subtrees of the return value p_{ret} and exception value p_{exc} root nodes are removed. In contrast to method parameters and static fields the return and exception values are created inside the method and are only used to pass computed values to the outside. There is no way for them to be used to pass information into the current method. The remaining non-root nodes are used as starting point for the ref-tree computation. Similar to the case of *mod*-tree extraction the computation adds each non-root node marked as referenced and all nodes on the path to its root node to the ref-tree as in Figure 8.

By now the object tree computation is finished and we can build the parameter passing interface for each method from the resulting *mod*- and ref-trees. All nodes from the ref-tree of a certain method are used as additional input parameters as well as all nodes from the *mod*-tree are used as additional output parameters. Figure 9 shows the code of the example program from Figure 3 including all additional input and output parameters.

```
1   static B foo(int x, B b)
2       IN:  b.x, b.a
3       OUT: b.a, b.a.i, p_ret.a, p_ret.a.i
4   {
5     A a = b.a;
6     int i = b.x;
7     {p_ret.a.i, b.a.i} = i + x;
8     return b;
9   }
10
11  static B bar()
12      IN:  B.s, B.s.x, B.s.a
13      OUT: B.s, B.s.a, B.s.a.i, p_ret.a, p_ret.a.i
14  {
15    B b = foo(23, s, IN: B.s.x, B.s.a,
16                     OUT: B.s.a, B.s.a.i, p_ret.a, p_ret.a.i);
17    return b;
18  }
```

Fig. 9. Methods from the example program from Figure 3 after side-effect computation. The code shows the additional parameters that have been created.

Blowup on may-aliasing situations The computation of side-effect using the object tree algorithm has the advantage that it does not only cover side-effects in form of abstract parameters. Additionally all parameters belong to a node in the corresponding object tree and therefore the parent and child nodes of the parameter can be determined. This is useful to see through which object fields the parameter in question could have been accessed. By following the path from the parameter node upwards through the parent nodes until a root node is reached, one knows which parameter, static variable or return value caused the propagation of the side-effect in the first place.

The drawback is that for each different access path of a single field like i in method *bar* of the example in Figure 6 a corresponding path in the object tree is created. This results in multiple nodes in the tree that belong to the same field. As node p_{ret} is a may-alias of node $B.s$, each field that may be accessed through the static field $B.s$ may also be accessed through the return value p_{ret} of method *bar*. Subsequently all children of one may-aliased node have to be children of all possible may-aliased nodes. Thus the more may-aliasing nodes are in the object tree, the more additional nodes have to be created.

The number of may-aliasing situations is directly related with the precision of the points-to analysis being used. The more precise the points-to analysis is, the less may-aliasing situations are found and hence less nodes have to be copied in the object tree. This is not a big issue for small programs, but analyzing larger programs is inherently difficult. In order to minimize the number of copies in the object tree, the number of may-aliasing situations has to be minimized

and therefore a precise points-to analysis has to be used. But precise points-to analyses are expensive and one may not be able to analyze a large program as precise as desired. This is a dilemma.

On the one hand an imprecise points-to analysis is quite fast, but it probably leads to an expensive object tree computation due to many may-aliasing situations. On the other hand a precise points-to analysis would probably lead to a faster object tree computation, but it is itself quite expensive. In both ways the side-effect analysis using object trees is going to be expensive.

The precision of points-to analysis is not solely the reason for may-aliasing situations. It is absolutely legal to have two or more object fields in a program instance referring to the same object, thus regardless of the employed points-to analysis may-alias situations may appear. Hence a solution that deals with may-aliasing situations more efficient has to be found in order to improve the scalability of the side-effect analysis. In the upcoming section object graphs are introduced. They improve the object tree computation and scale far better for larger programs or imprecise points-to information. In contrast to object tree computation the object graph computation is faster when an imprecise points-to analysis is used. This behavior alone allows us to increase the maximum size of programs that can be analyzed by factor 2 to 6.

2.3 Object graphs

Object trees as described in the previous section model the interprocedural side-effects in form of a tree where each node represents a set of object instances of the same type. They mimic the structure of object fields in natural way, but due to the alternating fixed point computations during the intra- and interprocedural propagation phases, they lack efficiency. Additionally their scalability heavily depends on the number of may-aliasing situations found in the program and thus on the precision of the points-to analysis used.

Our goal was to create a side-effect model utilizing the advantages of object trees in terms of precision whilst improving scalability especially for programs including many may-aliasing situations. Object graphs eliminate the need to copy nodes and whole subtrees in alias situations and therefore scale better for programs analyzed with inexpensive thus imprecise points-to analyses. They also avoid the mutually alternating fixed point computation steps of the inter- and intraprocedural parts and turn the interprocedural step into a much cheaper data flow analysis problem.

Algorithm In order to prevent copying whole subtrees in alias situations, subtrees are shared among multiple parent nodes. So object graph nodes are allowed to have more then a single predecessor node. Graph nodes are very similar to candidates for tree nodes from the object tree computation. They consist of a tuple with four entries: A points-to set *basePts* pointing to objects whose field $field$ may have been accessed. The type of field access is stored in *access*. Such an access may be referencing (ref) or modifying (mod) a certain field or both.

The entry *pts* is the points-to set of the locations the field *field* may be pointing to.

$$GraphNode := (basePts, field, pts, access) :$$
$$Points\text{-}to\ set \times Field \times Points\text{-}to\ set \times \{mod, ref, \{mod, ref\}\}$$

As a further optimization their successor and predecessor relations are not directly stored. They are inferred dynamically from their points-to sets, field names and object types. An object graph node n_1 is successor of another object graph node n_2 iff $n_2.pts$ is may-aliasing $n_1.basePts$ and $n_2.field$ may refer to a class that has a field matching type and name of $n_1.field$.

The computation algorithm of object graphs can be divided into 5 non-alternating main parts including an optional refinement step that improves the precision. A detailed description in form of pseudocode is available in the appendix B.2. The 5 main parts are:

1. Determine the root nodes by using the method signature
2. Build intraprocedural nodes for each heap access operation in the current method
3. Propagate side-effects interprocedurally from callee to caller
4. Refine computed side-effects by applying an escape analysis (optional)
5. Extract separate *mod* and *ref* sets and create additional method parameters from set elements.

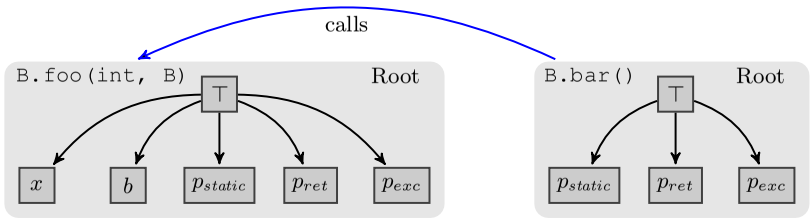

Fig. 10. First step of object graph computation for the example in Figure 3 showing the created root nodes.

The first phase determines the set of parameter and return values that may initially be used to pass information into the method or from the method to the outside. During this phase each method is processed exactly once. The root set of object graph nodes is created from the method signature in a very similar way to the object tree computation. Nodes are created for each parameter as well as for the return and exception values. The root graph for the example program of Figure 3 is shown in Figure 10.

After the root set has been created each instruction is processed exactly once and an object graph node is created iff the instruction accesses data on the heap.

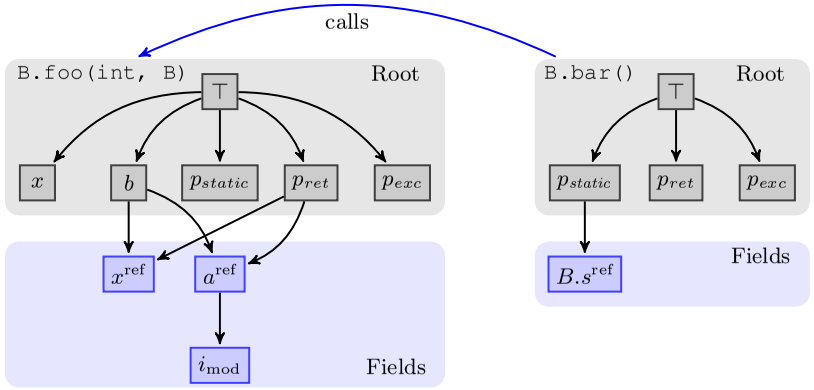

Fig. 11. Object graph for the example in Figure 3 after the intraprocedural phase.

In the example program of Figure 3 this has to be done for two methods: *bar* and *foo*.

Method *foo* contains 3 field accessing operations in lines 4, 7 and 13 of the byte code. For each operation a matching graph node is created and added to the current object graph. The `getfield` instruction in line 4 accesses the field a of the object in parameter b. As *foo* uses parameter b as return value, p_{ret} is referring to the same location as b. So the predecessor of the new node a is not only b but also p_{ret}. The next `getfield` instruction in line 7 refers to field x of parameter b. So a new node x is created and again b and p_{ret} are both predecessors of x. The last field access in *foo* is a `setfield` instruction in line 13. This instruction modifies the value of the field i. The base object of this field is stored in a local variable and refers to the same location as the field a that has been referenced by the instruction in line 4. So a is a predecessor of i.

Method *bar* executes only a single field access through `getstatic` in line 20 of the Bytecode. This operation references the static field $B.s$. So a new node for $B.s$ is created. As the operation accesses a static field, the root node p_{static} qualifies as predecessor.

By now all instructions potentially leading to side-effects have been processed once and the intraprocedural step is finished. The final result of this step is shown in Figure 11.

In the third phase the determined side-effects are propagated interprocedurally from callee to caller. We create candidates for each field node of the callee object graph and propagate them to all possible callers of the current method. As long as a caller doesn't already have a matching field node for the candidate, the algorithm creates a new field node.

In the current example, we have to propagate the side-effects of method *foo* to method *bar*. We create candidates for all 3 field nodes of *foo* and check if

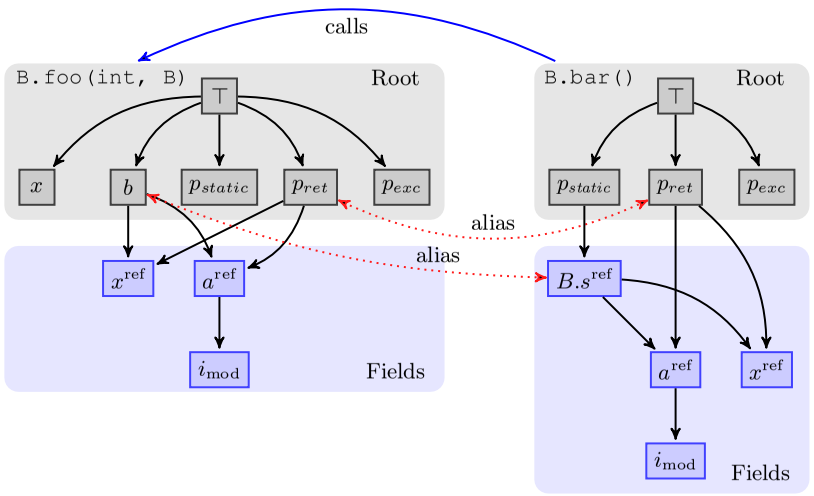

Fig. 12. Object graph for the example in Figure 3 after the interprocedural propagation.

method *bar* already has a field node matching one of the candidates. The first field node x has no matching counterpart in the object graph of method *bar*. So a new field node x is created. The same holds for the other two field nodes a and i of method foo and new field nodes in method *bar* are created for them, too. Method *bar* calls method foo using the static field $B.s$ as parameter b of foo. So $B.s$ and b are referring to the same location and consequently all field nodes reachable from b may also be reachable from $B.s$. As b is predecessor of x and a in the object graph of method foo, $B.s$ is going to be the predecessor of x and a in the object graph of method *bar*. The same reasoning can be used for the return values of method foo and *bar*, because *bar* uses the return value of foo as its own return value. So each successor of p_{ret} in foo that has a counterpart in *bar* is also the successor of p_{ret} in method *bar*. A quick look at Figure 11 shows that fields x and a are successors of p_{ret}. All potential side-effects of method foo have been processed by now and the interprocedural phase is finished. The resulting object graphs can be seen in Figure 12.

The overall number of candidates is already determined by the field nodes that were created in the intraprocedural step, because in the interprocedural phase new field nodes are created from existing candidates. So this new field nodes do not lead to new candidates. This observation is crucial, because it allows us to compute the interprocedural propagation very efficient as a monotone data flow problem [10]. A detailed description of the data flow problem for interprocedural propagation of side-effects is described in the appendix in section A. This description includes also an instantiation of the data flow problem for the current example program.

In contrast to object tree construction, object graph nodes are added no matter if the current graph contains matching predecessor nodes or not. This allows us to skip the alternating computations of the intraprocedural and interprocedural phase, but it is less precise as the result of the object tree computation. The object graphs computed by the third phase are already usable but they conservatively approximate the possible side-effects, which can be improved. These object graphs may contain nodes referring to field accesses that will never be visible outside the scope of the method they belong to. These nodes can be found by searching through their predecessors in order to find a path to a node of the root set. Only if an access path from a root node to the field node exists, its value may ever be reached from outside the method itself. Otherwise the effects of the field access cannot escape the current method and can be removed from the side effect analysis. The object graphs of the current example do not contain any non escaping nodes, as a path to a root node exists for all field nodes. So the result of the refinement stays the same as in Figure 12. A further example in Figure 16 is used to explain these effects in detail in the upcoming subsection.

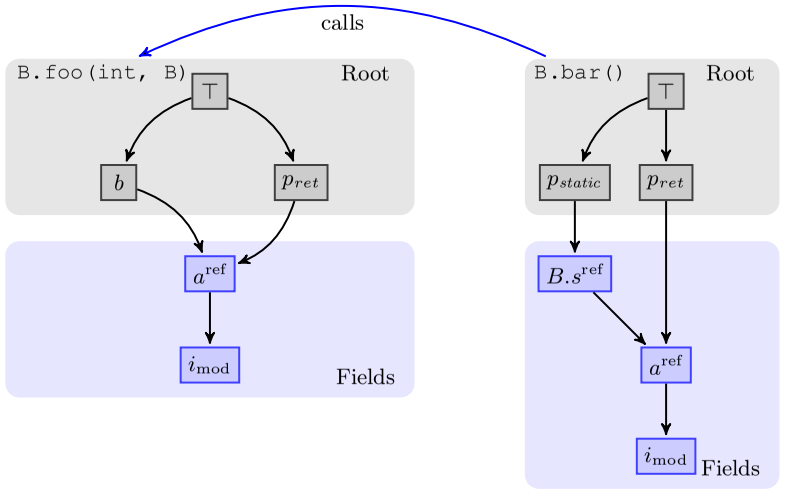

Fig. 13. Object graphs for the example program in Figure 3 containing only nodes referring to modified object instances.

By now all possible side-effects are already included in the object graph structure. In the last phase the *mod*- and *ref*-graphs are extracted from the object graph and used to create the additional input and output parameters for each method. The extraction works as in the object tree algorithm. The *mod*-graph should contain all modified fields as well as all objects who may contain a field that has been modified. So we start with all nodes marked as modified

mod and subsequently add all nodes that are predecessors of the nodes in the current set until the current set does not change anymore. Predecessor nodes are added because if a node n is a predecessor of node c it means that n contains a field that may refer to c. And if c has potentially been modified so has n. In the example from Figure 12 the computation of the *mod*-graph of method foo starts with the only node marked as modified, node i. Then all predecessors of i are collected and also added. The same happens for method *bar* where also the single node i is marked as modified and thus used as starting point. The *mod*-graphs with both i nodes and all their predecessors is shown in Figure 13.

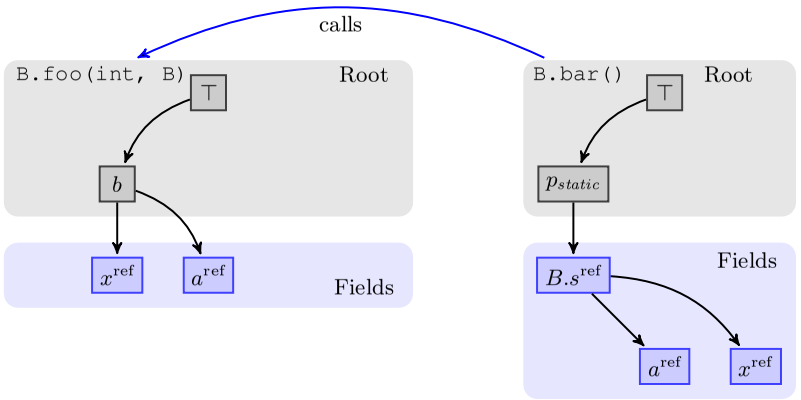

Fig. 14. Object graphs for the example program in Figure 3 containing only nodes belonging to referenced object instances.

In order to get all nodes that refer to values that may have been read the *ref*-graph is extracted from the object graph. The elements of the resulting *ref*-graph will later be used to create additional input parameters, so we are only interested in nodes that represent data that may have been passed into the current method. This holds definitely not for the return and exception values, because they are created inside the method and used to pass data to the outside. So p_{ret} and p_{exc} are cannot be part of the *ref*-graph. We start *ref*-graph extraction with all nodes of the root node set except p_{ret} and p_{exc}. All nodes that are reachable from the current set of nodes and that are marked are referenced (*ref*) are also included in the *ref*-graph. When a root node has no successors it may be left out. In the current example in method foo we notice that root node for field b has two successor nodes x and a, both marked as referenced. So these nodes are added to the *ref*-graph of foo. Field a has field i as successor, but as i is only marked as modified and not as referenced, i is not included in the *ref*-graph. Method *bar* has three field nodes marked as referenced: $B.s$, a and x. As in

method foo the field node i is left out because it is only marked as modified. The resulting *ref*-graphs are shown in Figure 14.

```
1   static B foo(int x, B b)
2       IN:  b.x, b.a
3       OUT: {b, p_ret}.a, {b, p_ret}.a.i
4   {
5     A a = b.a;
6     int i = b.x;
7     {b, p_ret}.a.i = i + x;
8     return b;
9   }
10
11  static B bar()
12      IN:  B.s, B.s.x, B.s.a
13      OUT: B.s, {B.s, p_ret}.a, {B.s, p_ret}.a.i
14  {
15    B b = foo(23, s, IN: B.s.x, B.s.a,
16                     OUT: {B.s, p_ret}.a, {B.s, p_ret}.a.i);
17    return b;
18  }
```

Fig. 15. Methods from the example program from Figure 3 after side-effect computation with object graphs. The code shows the additional parameters that have been created.

In the last step the existing methods are equipped with additional parameters that model all possible side-effects. These methods are now side-effect free, as all their in-coming and out-going data is captured through parameter passing. The methods of the example in Figure 3 convert into the methods of Figure 15. In contrast to the code originating from the object tree algorithm in Figure 9, the code originating from object graph computation contains less additional parameters. These additional parameters were the ones that emerge from the nodes of a subtree that has been copied due to an alias situation. This shows how object graphs can help to reduce the number of additional parameters.

Enhance precision with escape analysis The fourth phase during object graph computation is the optional refinement through escape analysis. In the previous example from Figure 3 this phase had no impact on the object graphs. In general this is not the case and evaluation shows that the refinement phase leads to a precision gain of about 10-15%. We are going to illustrate this effect in the following example.

Consider the program in Figure 16 where an object `A` is created as local variable and never passed to the outside of method `noEscape`. Evaluation shows

```
1  public class A {
2
3    public int i;
4
5    public static void noEscape() {
6      A a = new A();
7      a.i = 42;
8    }
9
10 }
```

Fig. 16. The modification to the field `a.i` will never be visible to callers of the method because there is no way of accessing the object created inside `noEscape`

that these kind of non-escaping field access operations appear to be common in real world programs.

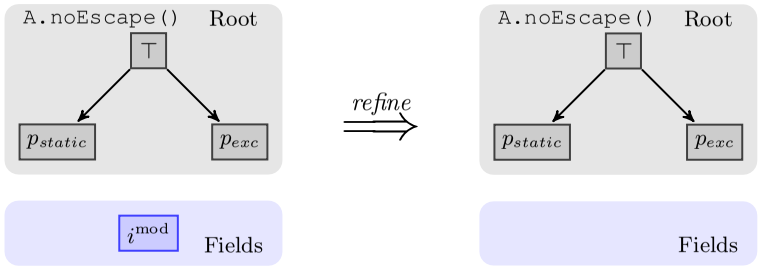

Fig. 17. Object graph and refined object graph of the program in Figure 16. The refined graph is on the right side.

At first we examine the result of the object graph computation for method *noEscape* without applying the optional refinement. As usual the root nodes are created from the method signature. Since the method doesn't have a return value and doesn't use parameters, the set of root nodes is quite small and contains only the node for static field accesses p_{static} and the node for exception values p_{exc}. During the intraprocedural analysis a single heap accessing operation `setfield` in line 7 is found. This operation modifies the field i of the local variable a which points to an object of type A. So a candidate is created for this operation and a matching node is added to the graph. Note that no root node qualifies as potential base object for the field access, because the access to field i is neither a static field access nor does it use an exception as base object. Therefore the new node for i has no predecessors.

Method *noEscape* does not contain any calls[3], so the result of the interprocedural propagation is the same as the result of the intraprocedural phase. The left side of Figure 17 shows the object graph for method *noEscape* that has been computed without the optional refinement phase.

When the refinement is applied all field nodes that are not on a path to a root node are removed. As the node for field i has no predecessors, it definitely does not lie on such a path. So this node is removed and the resulting graph does not contain anymore field nodes. As consequence no additional parameter has to be created for method *noEscape* and thus it is correctly detected that this method does not introduce any side-effects.

In general the evaluation of larger programs shows that those non-escaping nodes may introduce additional dependencies and therefore have a negative effect on the overall precision of the system dependence graph. They should always be removed as far as the precision of the points-to analysis allows us to definitely decide that they may not escape. Nevertheless, this step is optional and the object graph still remains a correct and conservative approximation of the method side-effects without the refinement phase.

3 Evaluation

We evaluated the performance and precision of SDG creation on 22 Java programs with up to 4 different points-to analyses and three different parameter passing models. The basic approach using a single parameter per instruction (section 2.1), the object tree (section 2.2) and object graph (section 2.3) model for side-effect computation. The implementation is written in Java and is based on the publicly available WALA program analysis framework[8]. The four points-to analyses we used can be characterized as follows.

1. Context insensitive type based: Disambiguating objects according to concrete types
2. Context insensitive instance based: Disambiguating objects according to instantiation sites
3. Context insensitive instance based with context sensitivity for Container classes: Unlimited context sensitivity for methods of objects implementing the Java Collection interface
4. Object sensitive instance based: Object sensitivity for all instantiation sites

Additionally our old SDG creation utility using a context insensitive instance based points-to analysis in combination with an object tree model, has been tested against 4 JavaCard programs of the set of example programs.

The programs were analyzed including all library methods they referenced and native methods were conservatively approximated through hand written method stubs that were analyzed instead. This is a huge difference to other approaches where parts of the runtime library are not analyzed at all [28]. As

[3] We ignore constructor calls in order to keep the example simple

the Java runtime library is quite big, those parts tend also to be big and have an enormous impact on the runtime of the analysis. But when left out, the result of the analysis is no longer a conservative approximation.

All tests were executed on an computer with 8x Dual-Core AMD Opteron(tm) 8220 processors and 32GB of RAM running Ubuntu Linux 7.10 with Java 6 64-bit. The implementation itself was designed as a single threaded application and used only one processor at a time.

Program name	LoC	Library	Joana	Object tree				Object graph				Wala			
				1.	2.	3.	4.	1.	2.	3.	4.	1.	2.	3.	4.
CorporateCard	485	JavaCard	√	√	√	√	√	√	√	√	√	√	√	√	√
Purse	5066	JavaCard	√	√	√	√	√	√	√	√	√	√	√	√	√
Wallet	124	JavaCard	√	√	√	√	√	√	√	√	√	√	√	√	√
Safe	577	JavaCard	√	√	√	√	√	√	√	√	√	√	√	√	√
Barcode	3462	J2ME 2.0	—	———				√	√	——		———			
bExplore	5913	J2ME 2.0	—	———				√	√	——		———			
J2MESafe	2237	J2ME 2.0	—	———				√	√	——		———			
KeePassJ2ME	4984	J2ME 2.0	—	———				√	√	——		———			
OneTimePass	1281	J2ME 2.0	—	———				√	√	——		———			
Battleship	330	JRE 1.4	—	√	√	√	√	√	√	√	√	√	√	√	√
JavaGrande (11 Programs)	4556	JRE 1.4	—	√	√	√	—	√	√	√	—	√	———		
HSQLDB	63304	JRE 1.4	—	———				√	√	——		———			

Table 2. Overview of all programs and configurations analyzed

Table 2 shows an overview of all programs analyzed. The first three columns contain the program name, the lines of code of the Java source and the library used. The evaluated programs use one of three different libraries. Corporate Card, Purse, Wallet and Safe are written for the relatively small JavaCard[4] library containing about 500 lines of code. Barcode, bExplore, J2MESafe, KeePassJ2ME and OneTimePass use the J2ME library[5] with about 30.000 lines of code. The rest of the programs use the standard Java 1.4 runtime library, whose size is about 100.000 lines of code. Those programs are: Battleship is a small implementation of the well known known game. The next 11 programs are part of the JavaGrande benchmarking suite [19] and share the same code base. And finally the biggest program is HSQLDB [27], a relational database engine written in Java.

The last four columns of Table 2 show the kind of side-effect model and points-to analysis that has been used during evaluation. The column named *Joana* refers to our old implementation which is based on the Harpoon Framework [22] and uses object trees and a context insensitive instance based points-to

[4] JavaCard is built for applications that run on smart cards and other devices with very limited memory and processing capabilities.

[5] J2ME is a Java environment for mobile phones

analysis. Columns *Object tree*, *Object graph* and *Wala* refer to the current implementation and are further divided into 4 sub columns numbered from *1.* to *4.* denoting the kind of points-to analysis used. The basic parameter model using a new parameter for each instruction is named *WALA*, because it is the default approach of the SDG that is part of the WALA framework. A "√" denotes that the program could be analyzed with the corresponding parameter model and points-to analysis, while a "—" denotes that an analysis was not possible, because it took too much space and time to compute.

The non-alternating intra- and interprocedural parts of the object graph computation as well as their formalization as a data flow analysis problem promise to improve the runtime of the side-effect analysis when compared to the object tree algorithm. In the next subsection we are going to evaluate if the predicted improvement in execution time is in fact present.

3.1 Execution time

Side-effect computation is an integral part of system dependence graph computation but it is not the only part. Other parts like the summary edge computation and the interprocedural data flow analysis use the results of the side-effect computation as input to their own computations. Therefore it is not sufficient to say that a faster side-effect computation will always lead to a faster system dependence graph computation, because the parts following side-effect computation may be slowed down due to the structure and size of their input. In order to see the effects of the different side-effect computation approaches we measured the execution time of system dependence graph creation in seconds for the programs displayed in Table 2.

Figure 18 to Figure 20 show the results of the execution time evaluation. Figure 18 contains the cumulated execution times for the SDG computation of all JavaCard programs. They include the side-effect computation, but the subsequent summary edge computation is not incorporated into this chart. The time needed for summary edge computation is shown in a separate chart on Figure 19 to make the effects of the parameter model more visible. The time for summary edge computation is treated separately, because it is a phase that follows the computation of the parameter passing model and its computation time depends heavily on the result of this previous phase. Finally the chart on Figure 20 shows the combined total amount of time needed to create an SDG with summary edges. The charts are split into 3 sections, one section for each parameter passing model: The basic model using one node per instruction is named *Wala*, the object tree model is labeled *Tree* and the object graph model is named *Graph*. Each section shows 4 bars, where each bar denotes the result under a particular points-to analysis. The context insensitive type based analysis is labeled *Type based*, the context insensitive instance based analysis is labeled *Instance based*, the context insensitive instance based analysis with context sensitivity for container classes is labeled *Context for container* and finally the object sensitive instance based analysis is labeled *Object sensitive*.

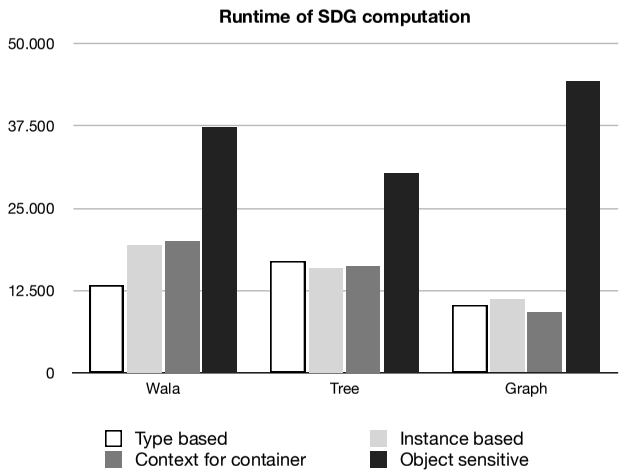

Fig. 18. Cumulated execution time in milliseconds of the system dependence graph creation for the JavaCard examples.

The execution time of the SDG computation without summary edges in Figure 18 consists mainly of the time of the call graph and points-to analysis construction and the following computation of the side-effects. The basic parameter model in the left section shows notable differences between the type based and instance based as well as between the instance based and object sensitive points-to analysis, despite the number and computation of additional parameters should be the same in all cases. The first gap between the type based and instance based approach can be explained through the additional time the more precise instance based points-to analysis needs to compute. This is also a reason for the bigger gap between instance based and object sensitive points-to analysis, but its not the only reason. The WALA framework implements context sensitivity through method cloning. There may be multiple instances of the same method, if they are executed in a different context. So context sensitivity increases the number of method instances and thus also the number of additional parameter nodes. This leads also to an extended runtime of the parameter model computation. In most cases the object graph parameter model is computed in the shortest amount of time. Only with the object sensitive points-to analysis the computation is taking longer then the object tree and the basic parameter model. Theoretically the basic parameter model should be the fastest algorithm, but in fact it is taking quite long to build. The number of parameter nodes that have to be created is almost twice as much as for the object graph model. This seems to slow down the computation. Another reason is that the implementation of this

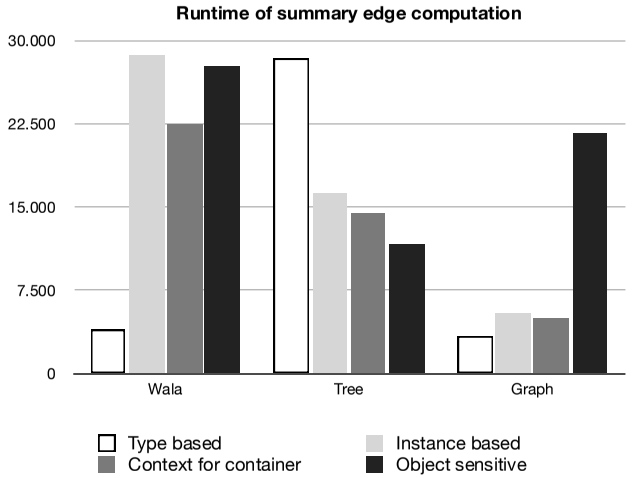

Fig. 19. Cumulated execution time in milliseconds of the summary edge computation for the JavaCard examples.

parameter model is taken directly from the Wala framework and may lack some performance tweaks. However in the case of the imprecise type based points-to analysis the basic model is nevertheless faster than the object tree model, that is slowed down by many aliasing situations. As long as the object sensitive points-to analysis is not used, the object graph model is the model to choose for fast computation times.

But the Evaluation showed also that in some cases it is not sufficient to use the parameter model that can be computed in the fastest way, because its result may slow down the following analyses. We observed this scenario for example when the basic parameter per instruction model is combined with the context insensitive instance based points-to analysis. Where the computation of the SDG without summary edges has been almost as fast as the computation using the less precise class based points-to analysis, but the summary edge computation, see Figure 19, took almost eight times longer, although the number of generated parameter nodes is almost identical. These kind of effects are hard to foresee and show that the runtime of the summary edge computation is not determined alone by the number of parameter nodes. There may also be effects that take a huge number of iterations to propagate through the whole graph and slow down the fixed point computation. The chart in Figure 20 supports the previous remarks that object tree computation scales worse for imprecise points-to analyses, like the context free class based or instance based analyses.

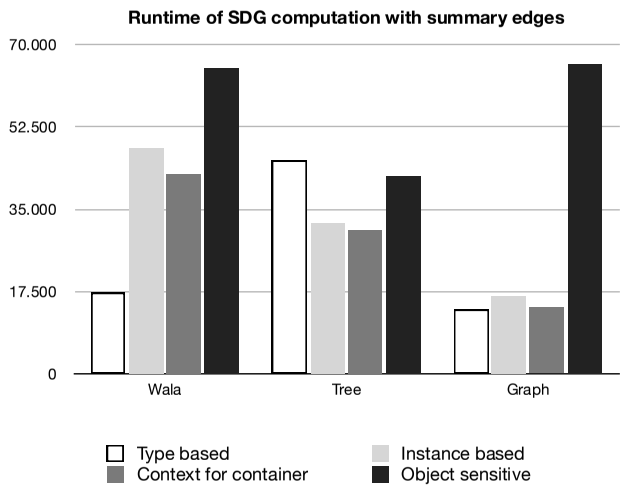

Fig. 20. Cumulated execution time in milliseconds of the system dependence graph creation and summary edge computation. Four different points-to analyses from simple context insensitive class based to object sensitive instance based show the effect of points-to analysis precision on the execution time.

In general we observed a huge difference in the execution time of the analysis while the precision of the resulting graphs did only sightly improve [6]. In most cases the object tree and also the basic one node per instruction parameter model lead to a runtime that is 2 to 10 times slower then the object graph model. But as Figure 19 shows, the execution time of the summary edge computation for graphs using the object tree model is steadily decreasing when the precision of the points-to analysis increases. This is because the size of the object trees and thus the number of additional parameters decrease when the points-to analysis is more precise, as less may-aliasing situations appear. So in Figure 20 a turnaround finally occurs when the object sensitive points-to analysis is used and the object tree model outperforms the object graph model The increased runtime of the graph based approach can be explained through the huge number of field nodes created, leading to a much longer runtime of the summary edge computation[7]. The more distinguishable points-to sets exist the more field nodes are potentially created in the object graph approach while the object tree approach only supports one child node per field and simply merges field nodes that have the same parent but different points-to sets.

[6] The results of the upcoming section on precision support this statement

[7] summary edge computation has a asymptotic running time of $O(Nodes^3)$ [21]

3.2 Precision

In the previous subsection we showed how the runtime of system dependence graph creation is affected by different parameter models and through various points-to analyses. This section focuses on the question if a longer runtime automatically results in an equal large gain in precision and which configuration provides the best trade off between short runtime and high precision. Therefore we measured the gain or loss of precision of the system dependence graphs that have been created under various configurations.

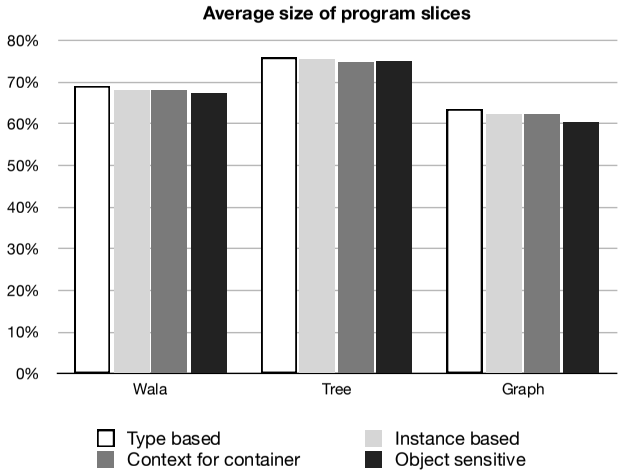

Fig. 21. Average size of program slices for the JavaCard examples under various points-to and side-effect configurations.

One way to measure the precision of a system dependence graph is by counting the average number of nodes contained in a *program slice* [2,18,14]. A program slice contains a subset of all nodes of the system dependence graph. Those nodes represent program statements that may be influenced by the so called *slicing criterion*. The slicing criterion itself is also a set of nodes of the system dependence graph. The computation of those slices is a very common application for system dependence graphs and has been explained in detail in various publications [7,20,25,13,11].

A slice is a conservative approximation of all statements that potentially influence the slicing criterion and it is computed directly from the system dependence graph. The more precise the dependency graph is the less nodes are contained in average in a slice. More nodes per slice reads as more possible de-

pendencies and therefore less precision, so a low average is desirable. In order to compute an average that can be compared among the different parameter models only nodes belonging to concrete program statements were considered relevant and especially no parameter nodes were taken into account. Each relevant node is used as a discrete slicing criterion and a program slice is computed using the context sensitive two phase slicing algorithm from Horwitz, Reps and Binkley [7]. The precision is displayed as percentage of the average number of relevant nodes in those slices compared to the total number of relevant nodes in the system dependence graph.

Figure 21 shows the average precision of the system dependence graphs of all *JavaCard* programs from the set of example programs on Figure 2. These programs were analyzed with all available points-to analysis and side-effect models. The chart is grouped into three regions, where the left region contains all results using basic parameter model, the middle region contains the results of the object tree model and the right region contains all results using the object graph model. For each points-to analysis a different bar color is used. Our old SDG implementation that uses a context free instance based points to analyses and the object tree side-effect model shows an average precision of about 61% for the JavaCard test cases. This percentage seems to be quite good when compared to the precision of the object tree model of the current implementation, but they cannot be directly compared as the old implementation uses a different intermediate representation that is more verbose in certain parts. Constant declarations for example are explicitly present as instructions in the intermediate representation of the Harpoon Framework used by the old implementation. The intermediate representation of the Wala framework has an integrated constant propagation and does not contain instructions for constant field declarations.

The results of the system dependence graphs using the object tree computation are about 10% less precise then the ones using object graphs while the precision of the basic model is in between. In general the difference between the parameter models is by far greater then the difference between the points-to analyses. This may be explained through the nature of the JavaCard programs. As the JavaCard programs and the JavaCard library are though to run on tiny smart cards with only limited resources, they try to save memory and create new objects very rarely. Because only few objects are used in those programs, the precision of the points-to analysis has no huge effect on the overall precision of the system dependence graph. However a gain of 3-5% from the imprecise type based context free approach to the more precise instance based object sensitive approach is still present.

For the same reason, almost no difference in precision is found when the optional refinement phase of the object graph model is left out. Only few objects are used inside JavaCard programs and even less are used only for internal computations, so the refinement has almost no effect. But this is not always the case. For larger programs like the examples of the JavaGrande benchmark suite the effect of the refinement phase and the effect of different points-to analyses is very visible. In Figure 22 the differences between the *Graph (no refine)* region

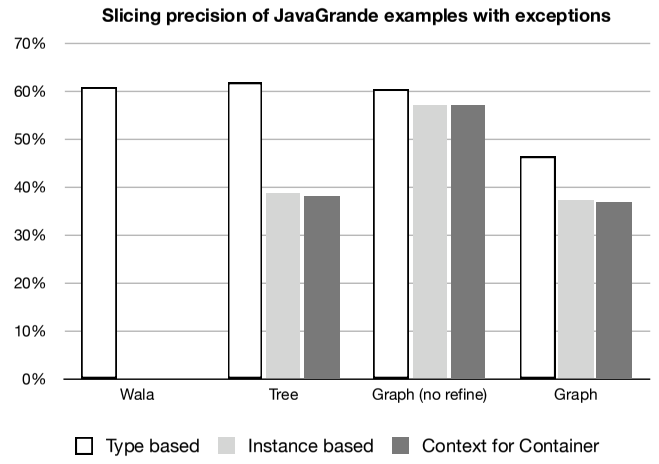

Fig. 22. Average size of program slices of the examples from the JavaGrande benchmark suite.

in the middle and the *Graph* region on the right show a gain of almost 20% in precision for object graphs with the optional refinement and the instance based points-to analysis. For a less precise points-to analysis like the type based approach the gain is still quite big but with about 14% it is not as huge as for the more precise points-to analyses. This effect is even bigger for the object tree approach, which applies an escape analysis implicitly during computation. In the *Tree* region on the left side of the chart the difference between the imprecise type based and the more precise instance based points-to analyses is about 24%. This clearly shows that the escape analysis that is implicitly used by object trees and explicitly used by object graphs depend largely on the precision of the points-to analysis.

The largest programs of the evaluation are HSQLDB and the programs using the JavaME framework. Because of their size we were only able to compute a system dependence graph for them using the object graph model and the two least precise points-to analyses. The average program slice of SDGs created with the type based context free points-to analysis contains about 71% of all relevant nodes and the about 68% for the more precise instance based context free points-to analysis. The difference between the type based and the instance based context free analysis is a gain of about 3% in precision while the runtime almost doubled. More precise points-to analyses could not be used because the points-to computation itself did not finish in a reasonable time frame. With the less precise points-to analyses that worked for the object graph the object tree propagation consumed too much time and memory, so no results could be retrieved. The basic parameter model used by the Wala framework did also not

finish in time. It seems to create too much additional parameter nodes in order to be useful for the analysis of larger programs. This shows clearly that object graphs are the better choice when analyzing larger programs as they can deal with imprecise points-to analyses more efficiently.

In general it seems quite strange that more precise points-to analyses do not seem to have such a big effect on the overall precision of a system dependence graph. In case of the JavaGrande examples the move from type based to instance based points-to analysis did have an effect but only when it is combined with the application of an escape analysis it becomes more visible. The HSQLDB and the J2ME examples however only show a gain in precision of about 3% which is quite small when compared to the gain of up to 24% in the JavaGrande examples. This can be explained through the nature of the programs in the JavaGrande benchmark suite. Those programs compute mathematical problems. The computation mostly takes place inside a particular method and this method creates instances of helper classes that support the computation. But the references to those helper classes are never passed to the outside. The escape analysis is able to detect them when a sufficient precise points-to analysis is used and the effect is the quite huge gain in precision. The HSQLDB and J2ME examples however do not mainly run method local computations. They have a more global structure were data is passed and modified through larger parts of the program. They simply do not have as many non-escaping objects inside their methods and so the escape analysis does not detect as many as in the JavaGrande programs.

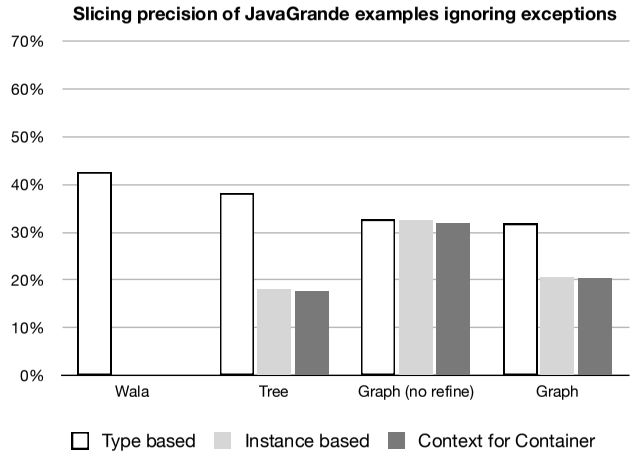

Fig. 23. Average size of program slices of the JavaGrande examples without considering the effects of exceptions.

While the effect of the different points-to analyses has been smaller than expected the evaluation shows that the influence of control flow and data flow through exceptions is by far greater. Figure 23 contains the chart for the JavaGrande examples that were analyzed without incorporating the effects of exceptions. This chart can be directly compared to the chart in Figure 22 that shows the results of the same programs including the effect of exceptions. In theory almost any Java Bytecode operation may throw an exception, even iff this is very rarely the case in practice. Especially all field accessing instructions potentially throw a NullPointerException whenever the referenced base object does not exist. In certain situations it is impossible for such an exception to occur, as the base object definitely exists. Simply consider two field-get operations that are executed directly one after the other and refer to the same base object. The second field-get operation can only be executed iff the base object exists, as the first field-get instruction otherwise would have already thrown a NullPointerException and the second operation would have never been executed. Our analysis currently ignores this kind of situations were exceptions can definitely not occur and instead use the conservative approximation that an exception may always be raised.

Comparing the results from Figure 23 to Figure 22 we see that the average size of the program slices without the effect of exceptions almost decreased to half of their original size. Because of this observation we strongly suggest that a more precise analysis of the effects of exceptions will be very beneficial to overall precision of the system dependence graph.

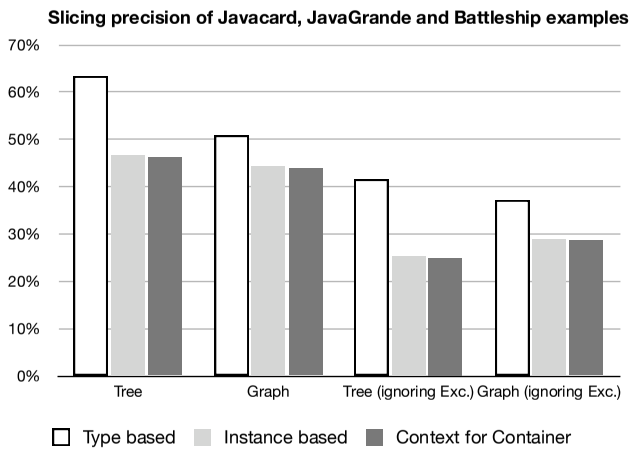

Fig. 24. Average size of program slices of the JavaCard, JavaGrande and Battleship examples with and without considering the effects of exceptions.

This suggestion is also supported by the combined evaluation of the Battleship program and all JavaCard and JavaGrande programs. Figure 24 shows the average size of a program slice for those programs with three different points-to analyses and using object trees as well as object graphs. The two regions on the left show the results under various options and include the effect of exceptions during the analysis while the other two regions on the right show the results for the same options without the effect of exceptions. Again the average slice size of the analyses ignoring exceptions is almost only half of the size of their matching counterparts that include the effects of exceptions.

4 Summary and Outlook

We have shown that side-effect computation has a huge impact on the precision and scalability of system dependence graph computation. Some approaches work better together with imprecise points-to analyses than others and this work provides a basis to choose the right side-effect analysis for the concrete situation. In general the object graph approach seems to deliver the best tradeoff between scalability and precision. But the object tree approach is the better choice whenever a very precise points-to analysis is used.

The precision gained through more precise points-to analyses has been smaller than expected in most cases. However the step from the imprecise type based analysis to the context free instance based analysis showed the biggest effects. The step from the context insensitive to the object sensitive analysis had only very little impact on the overall precision. We expect a significantly gain of precision from context sensitive points-to analyses only for larger programs. As of today the scalability of context sensitive points-to analyses and the side-effect computation models is limiting their application to small programs. Whaley and Lam showed that context sensitive points-to analysis is possible for larger programs[29], but the current bottleneck is the creation of the interprocedural interface for the parameter passing of the side-effects and the summary edge computation.

Interestingly while the impact of precise points-to analyses on the overall precision of the analysis seems to be quite small and is varying depending on the nature of the program, the effects of additional control and data flow through exceptions are always very visible. During evaluation precision almost doubled when the effects of exceptions were ignored. Of course an analysis ignoring those effects is no longer sound, but this observation supports the conclusion that the current treatment of exceptions in SDGs pollutes the precision of the overall result and should be investigated further.

References

1. Samuel Bates and Susan Horwitz. Incremental program testing using program dependence graphs. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 384–396, New York, NY, USA, 1993. ACM.

2. David Binkley, Mark Harman, and Jens Krinke. Empirical study of optimization techniques for massive slicing. *ACM Trans. Program. Lang. Syst.*, 30(1):3, 2007.
3. Matthew B. Dwyer and John Hatcliff. Slicing software for model construction. In *Higher-order and Symbolic Computation*, pages 105–118, 1999.
4. Christian Hammer and Gregor Snelting. An improved slicer for java. In *PASTE '04: Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 17–22, New York, NY, USA, 2004. ACM.
5. Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. publikation 2008-16, Fakultät für Informatik, Universität Karlsruhe (TH), Germany, November 2008. Supersedes ISSSE and ISoLA 2006.
6. John Hatcliff, James Corbett, Matthew Dwyer, Stefan Sokolowski, and Hongjun Zheng. A formal study of slicing for multi-threaded programs with jvm concurrency primitives. In *In Proceedings of the 6th International Static Analysis Symposium (SAS'99*, pages 1–18, 1999.
7. Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.
8. IBM. T.J. Watson Libraries for Analysis (WALA). http://wala.sf.net/.
9. Mariam Kamkar, Nahid Shahmehri, and Peter Fritzson. Bug localization by algorithmic debugging and program slicing. In *PLILP '90: Proceedings of the 2nd International Workshop on Programming Language Implementation and Logic Programming*, pages 60–74, London, UK, 1990. Springer-Verlag.
10. Gary A. Kildall. A unified approach to global program optimization. In *POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206, New York, NY, USA, 1973. ACM.
11. Jens Krinke. *Advanced Slicing of Sequential and Concurrent Programs*. publikation, Universität Passau, April 2003.
12. Loren Larsen and Mary Jean Harrold. Slicing object-oriented software. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 495–505, Washington, DC, USA, 1996. IEEE Computer Society.
13. Donglin Liang and Mary Jean Harrold. Slicing objects using system dependence graphs. In *ICSM*, pages 358–367, 1998.
14. Donglin Liang and Mary Jean Harrold. Efficient points-to analysis for whole-program analysis. In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 199–215, London, UK, 1999. Springer-Verlag.
15. Tim Lindholm and Frank Yellin. *Java(TM) Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, April 1999.
16. Ferenc Magyar and Ferenc Magyar. Static slicing of java programs. Technical report, University, 1996.
17. Brian A. Malloy, John D. Mcgregor, Anand Krishnaswamy, and Murali Medikonda. An extensible program representation for object-oriented software. *ACM SIGPLAN Notices*, 29:38–47, 1994.
18. Markus Mock, Darren C. Atkinson, Craig Chambers, and Susan J. Eggers. Improving program slicing with dynamic points-to data. *SIGSOFT Softw. Eng. Notes*, 27(6):71–80, 2002.
19. EPCC University of Edinburgh. The Java Grande benchmarking suite. http://www.epcc.ed.ac.uk/research/activities/java-grande/.

20. Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 177–184, New York, NY, USA, 1984. ACM.
21. Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. *SIGSOFT Softw. Eng. Notes*, 19(5):11–20, 1994.
22. Martin C. Rinard. The Flex Program Analysis and Compilation System. http://flex.cscott.net/Harpoon/.
23. Neil Walkinshaw Marc Roper. The java system dependence graph. In *In Third IEEE International Workshop on Source Code Analysis and Manipulation*, pages 5–5, 2003.
24. Manu Sridharan, Stephen J. Fink, and Rastislav Bodík. Thin slicing. In Jeanne Ferrante and Kathryn S. McKinley, editors, *PLDI*, pages 112–122. ACM, 2007.
25. Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.
26. Paolo Tonella, Giuliano Antoniol, Roberto Fiutem, Via Alla Cascata, and Ettore Merlo. Flow insensitive c++ pointers and polymorphism analysis and its application to slicing. In *In 19th International Conference on Software Engineering*, pages 433–443. IEEE Computer Society Press, 1997.
27. Fred Toussi. HSQL database engine. http://hsqldb.org/.
28. Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. Taj: effective taint analysis of web applications. *SIGPLAN Not.*, 44(6):87–97, 2009.
29. John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 131–144, New York, NY, USA, 2004. ACM.
30. Jianjun Zhao. Dependence analysis of java bytecode. In *COMPSAC '00: 24th International Computer Software and Applications Conference*, pages 486–491, Washington, DC, USA, 2000. IEEE Computer Society.

A Interprocedural propagation as data flow problem

A data flow problem consists of a graph, a lattice of values, a transfer function for each node of the graph and a join operation.

$$
\begin{aligned}
\mathit{CallGraph} &:= \{(m_1.m_2) \mid m_1, m_2 \in \mathit{Methods} \wedge m_1 \mathit{may\ call}\ m_2\} \\
\mathit{Candidates} &:= \{c \mid \exists m \in \mathit{Methods}, \\
&\qquad i \in \mathit{Instructions\ of}\ m : c\ \mathit{is\ candidate\ of}\ i\} \\
\mathit{in}(m) &= \bigcup \mathit{out}(m') \mid (m, m') \in \mathit{CallGraph} \\
\mathit{out}(m) &= f_m(\mathit{in}(m)) \\
\forall m \in \mathit{Methods} : f_m(X) &: \mathit{Candidates} \to \mathit{Candidates} \\
&= \mathit{gen}(m) \cup (X \setminus \mathit{kill}(m)) \\
\mathit{gen}(m) &: \mathit{Methods} \to \mathit{Candidates} \\
&= \{i \in \mathit{Instructions\ of}\ m : c\ \mathit{is\ candidate\ of}\ i\} \\
\mathit{kill}(m) &: \mathit{Methods} \to \mathit{Candidates} \\
&= \emptyset
\end{aligned}
$$

In the case of interprocedural propagation of side-effects for the object graph algorithm the graph used for the propagation of the data flow properties is the call graph. A standard call graph contains a node for each method in the program and edges between them iff a call may occur. The lattice of values is the powerset of all candidates with $\subseteq$ as relation and set unification is used as join operator. The transfer functions f_m are built as usual from the *kill* and *gen* sets of each method. Where the *gen* set contains all candidates that are created directly from the instructions of the current method and the *kill* sets stay empty.

The goal of the data flow analysis is to find a fixed point for the *in* and *out* functions of each method. This is achieved by setting the result of all *in* function initially to $\emptyset$ and then iterate the functions until the results of *in* and *out* do not change anymore.

A.1 Instantiation for the example program

For the example of Figure 3 the data flow problem instantiation is as follows.

$$
\begin{aligned}
\mathit{CallGraph} &= \{(bar, foo)\} \\
\mathit{Candidates} &= \{(pts(b), \text{'a'}, pts(b.a), ref), (pts(b), \text{'x'}, pts(b.x), ref), \\
&\qquad (pts(a), \text{'i'}, pts(a.i), mod), (pts(p_{static}), \text{'B.s'}, pts(B.s), ref)\} \\
\mathit{gen}(foo) &= \{(pts(b), \text{'a'}, pts(b.a), ref), (pts(b), \text{'x'}, pts(b.x), ref), \\
&\qquad (pts(a), \text{'i'}, pts(a.i), mod)\} \\
\mathit{gen}(bar) &= \{(pts(p_{static}), \text{'B.s'}, pts(B.s), ref)\}
\end{aligned}
$$

This leads to the transfer functions f_{foo} and f_{bar}.

$$\begin{aligned} f_{foo}(X) &= \{(pts(b),\ 'a', pts(b.a), ref), (pts(b),\ 'x', pts(b.x), ref), \\ &\quad (pts(a),\ 'i', pts(a.i), mod)\} \cup X \\ f_{bar}(X) &= \{(pts(p_{static}),\ 'B.s', pts(B.s), ref)\} \cup X \end{aligned}$$

The initial set-up of the fixed point computation for the *in* and *out* functions is:

$$\begin{aligned} in_1(foo) = in_1(bar) &= \emptyset \\ out_1(foo) &= f_{foo}(in_1(foo)) = f_{foo}(\emptyset) \\ &= \{(pts(b),\ 'a', pts(b.a), ref), (pts(b),\ 'x', pts(b.x), ref), \\ &\quad (pts(a),\ 'i', pts(a.i), mod)\} \\ out_1(bar) &= f_{bar}(in_1(bar)) = f_{bar}(\emptyset) \\ &= \{(pts(p_{static}),\ 'B.s', pts(B.s), ref)\} \end{aligned}$$

After the first iteration of the fixed point computation the results change.

$$\begin{aligned} in_2(foo) &= \bigcup out_1(m') \mid (foo, m') \in \{(bar, foo) = \emptyset = in_1(foo) \\ in_2(bar) &= \bigcup out_1(m') \mid (bar, m') \in \{(bar, foo) = out_1(foo) \\ &= \{(pts(b),\ 'a', pts(b.a), ref), (pts(b),\ 'x', pts(b.x), ref), \\ &\quad (pts(a),\ 'i', pts(a.i), mod)\} \\ out_2(foo) &= f_{foo}(in_2(foo)) = f_{foo}(\emptyset) = out_1(foo) \\ out_2(bar) &= f_{bar}(in_2(bar)) = \{(pts(p_{static}),\ 'B.s', pts(B.s), ref)\} \cup \\ &\quad \{(pts(b),\ 'a', pts(b.a), ref), (pts(b),\ 'x', pts(b.x), ref), \\ &\quad (pts(a),\ 'i', pts(a.i), mod)\} \end{aligned}$$

Another iteration reveals that the fixed point has been reached.

$$\begin{aligned} in_3(foo) &= \emptyset = in_2(foo) = in_1(foo) \\ in_3(bar) &= out_2(foo) = out_1(foo) = in_2(bar) \\ out_3(foo) &= f_{foo}(in_3(foo)) = f_{foo}(in_1(foo)) = out_1(foo) \\ out_3(bar) &= f_{bar}(in_3(bar)) = f_{bar}(in_2(bar)) = out_2(bar) \end{aligned}$$

All side-effects have been propagated and after creating the appropriate object graph field node for each candidate, the object graphs look like in Figure 12.

B Pseudocode

B.1 Object tree computation

In our implementation a node n consists of a link to its parent node $n.parent$, a field name $n.field$ and a points-to set $n.pts$ describing which locations the field may reference. The point-to set of the base object for field $n.field$ can be accessed through the parent object $n.parent.pts$.


```
procedure emit_root
IN: Method m
OUT: Objecttree t
begin
    t = empty Objecttree
    Add root node ⊤ to t
    for all parameters p of m do
        Add mod and ref node for p as child of ⊤ to t
    done
    Add p_static mod and ref node as child of ⊤ to t
    return t
end

procedure emit_from_instruction
IN: Instruction i
OUT: Set of emitted candidates candidates
begin
    if i is a field access then
        Create new candidate candidate
        if i is a static field access then
            candidate.basePts = p_static.pts
        else
            candidate.basePts = i.basePts
        fi
        candidate.field = i.field
        candidate.pts = i.pts
        if i is a field−get operation then
            candidate.access = ref
        else
            candidate.access = mod
        fi
        Add candidate to candidates
    fi
    return candidates
end

procedure emit_from_tree
IN: Objecttree t
OUT: Set of emitted candidates candidates
begin
    for all nodes n ∈ t do
        if n ≠ ⊤ and n.parent ≠ ⊤ then
            Create new candidate candidate
            candidate.basePts = n.parent.pts
            candidate.field = n.field
```

```
            candidate.pts = n.pts
            candidate.access = n.access
            Add candidate to candidates
        fi
    done
    return candidates
end

procedure coalesce
IN: Objecttree t, Set of emitted candidates candidates
OUT: Objecttree t'
begin
    t' = copy of t;
    for all candidates i ∈ candidates, ref−nodes p ∈ t do
        if p.pts is may−aliasing i.basePts and
        p.field may have a child field i.field and
        the unfolding criterion holds then
            if p has no child for field i.field then
                Add i as child of p to g'
            else
                Get child c of p for field i.field
                c.pts = c.pts ∪ i.pts
            fi
        fi
    done
    return t'
end

procedure adjust_local_interface
IN: Method m, Objecttree t
OUT: Objecttree t'
begin
    t' = copy of t
    repeat
        for all instructions i ∈ m do
            t' = coalesce(t', emit_from_instruction(i))
        done
    until t' does not change anymore
    return t'
end

procedure build_objecttree
IN: Method m
OUT: Objecttree t_m
begin
```



```
    for all methods m' reachable from m in the call graph do
        Objecttree t_m' = emit_root(m')
        t_m' = adjust_local_interface(m', t_m')
    done

    repeat
        for all methods m_1, m_2 reachable from m in the call graph do
            if m_1 → m_2 in call graph then
                t_m1 = coalesce(t_m1, emit_from_tree(t_m2))
                t_m1 = adjust_local_interface(t_m1)
            fi
        done
    until nothing changes anymore

    return t_m
end

procedure extract_mod−tree
IN: Objecttree t
OUT: Objecttree t_mod
begin
    t_mod = empty tree
    for all nodes f in mod−nodes of t do
        if f is field node then
            add f to t_mod
            add all nodes on the path from f to ⊤ in t to t_mod
        fi
    done
end

procedure extract_ref−tree
IN: Objecttree t
OUT: Objecttree t_ref
begin
    t_chop = copy of t
    remove p_ret, p_exc and all their children from t_chop
    t_ref = empty tree
    for all nodes f in ref−nodes of t_chop do
        if f is field node then
            add f to t_ref
            add all nodes on the path from f to ⊤ in t to t_ref
        fi
    done
end
```

B.2 Object graph computation

procedure emit_root
IN: Method m
OUT: Objectgraph g
begin
    g = empty Objectgraph
    Add root node $\top$ to g
    for all parameters p of m **do**
        Add mod and ref node of p with p.basePts = $\top$.pts to g
    done
    Add p_{static} mod and ref node with p_{static}.basePts = $\top$.pts to g
    return g
end

procedure emit_from_instruction
IN: Instruction i
OUT: Set of emitted nodes *candidates*
begin
    if i is a field access **then**
        Create new candidate *candidate*
        if i is a static field access **then**
            candidate.basePts = p_{static}.pts
        else
            candidate.basePts = i.basePts
        fi
        candidate.field = i.field
        candidate.pts = i.pts
        if i is a field−get operation **then**
            candidate.access = ref
        else
            candidate.access = mod
        fi

        Add *candidate* to *candidates*
    fi
    return *candidates*
end

procedure build_local_interface
IN: Method m, Objectgraph g^{root}
OUT: Objectgraph g
begin
    $g = g^{root}$
    for all instructions $i \in m$ **do**
        $g = g \cup$ emit_from_instruction(i)

    done
    return g
end

procedure prune_non_escaping
IN: Objectgraph g
OUT: Objectgraph g'
begin
    g' = empty Objectgraph
    repeat
        for all nodes $n \in g$ **do**
            if $n = \top$ **then**
                add n to g'
            else if $\exists n' \in g' : n'.pts \cap n$.basePts $\neq \emptyset$
                and n'.field may have a child field n.field
            then
                add n to g'
            fi
        done
    until g' does not change anymore
    return g'
end

procedure build_objectgraph
IN: Method m
OUT: Objectgraph g
begin
    for all methods m' reachable from m in the call graph **do**
        Objectgraph $g_{m'}^{root}$ = emit_root(m')
        Objectgraph $g_{m'}$ = build_local_interface(m', $g_{m'}^{root}$)
    done

    repeat
        for all methods m_1, m_2 reachable from m in the call graph **do**
            if $m_1 \rightarrow m_2$ in call graph **then**
                $g_{m_1} = g_{m_1} \cup (g_{m_2} \setminus g_{m_2}^{root})$
            fi
        done
    until nothing changes anymore

    g = prune_non_escaping(g_m)

    return g
end