Precision-Guided Context Sensitivity for Pointer Analysis

YUE LI, Aarhus University, Denmark
TIAN TAN, Aarhus University, Denmark
ANDERS MØLLER, Aarhus University, Denmark
YANNIS SMARAGDAKIS, University of Athens, Greece

Context sensitivity is an essential technique for ensuring high precision in Java pointer analyses. It has been observed that applying context sensitivity partially, only on a select subset of the methods, can improve the balance between analysis precision and speed. However, existing techniques are based on heuristics that do not provide much insight into what characterizes this method subset. In this work, we present a more principled approach for identifying precision-critical methods, based on general patterns of value flows that explain where most of the imprecision arises in context-insensitive pointer analysis. Accordingly, we provide an efficient algorithm to recognize these flow patterns in a given program and exploit them to yield good tradeoffs between analysis precision and speed.

Our experimental results on standard benchmark and real-world programs show that a pointer analysis that applies context sensitivity partially, only on the identified precision-critical methods, preserves effectively all (98.8%) of the precision of a highly-precise conventional context-sensitive pointer analysis (2-object-sensitive with a context-sensitive heap), with a substantial speedup (on average 3.4X, and up to 9.2X).

CCS Concepts: • **Theory of computation → Program analysis**;

Additional Key Words and Phrases: static analysis, points-to analysis, Java

ACM Reference Format:
Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Precision-Guided Context Sensitivity for Pointer Analysis. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 141 (November 2018), 29 pages. https://doi.org/10.1145/3276511

1 INTRODUCTION

Pointer analysis is a fundamental family of static analyses that estimate the possible values of pointer variables in a program. Such information is essential for reasoning about aliasing and inter-procedural control flow in object-oriented programs, and it is used in a wide range of software engineering tools, e.g., for bug detection [Chandra et al. 2009; Naik et al. 2006, 2009], security analysis [Arzt et al. 2014; Grech and Smaragdakis 2017; Livshits and Lam 2005], program verification [Fink et al. 2008; Pradel et al. 2012], and program debugging and understanding [Li et al. 2016; Sridharan et al. 2007].

For decades, numerous analysis techniques have been developed to make pointer analysis more precise and more efficient, especially for object-oriented languages [Hind 2001; Smaragdakis and Balatsouras 2015; Sridharan et al. 2013]. One of the most successful ideas for producing high precision is *context sensitivity* [Milanova et al. 2002, 2005; Sharir and Pnueli 1981; Shivers 1991; Smaragdakis et al. 2011], which allows each program method to be analyzed under different contexts,

Authors' email addresses: yueli@cs.au.dk, tiantan@cs.au.dk, amoeller@cs.au.dk, smaragd@di.uoa.gr.

2475-1421/2018/11-ART141
https://doi.org/10.1145/3276511

to separate the static abstractions of different dynamic instantiations of the method's variables and thereby reduce spurious object flows. However, despite great precision benefits, context sensitivity comes with heavy efficiency costs [Kastrinis and Smaragdakis 2013; Lhoták and Hendren 2006; Oh et al. 2014; Tan et al. 2016, 2017; Xu and Rountev 2008]. One reason is that, with conventional context-sensitivity techniques, every method in a program is treated the same, meaning that many methods that do not benefit from context sensitivity are analyzed for multiple contexts redundantly. As a consequence, too much space and time is consumed [Smaragdakis et al. 2014].

This naturally raises the question of whether it is possible to apply context sensitivity *selectively*, only for the methods where it is beneficial to the overall analysis precision. It is far from trivial to determine when a context-sensitive analysis will yield precision benefits (or conversely, to determine when omitting context sensitivity for a method would introduce imprecision). This challenge of effectively identifying the *precision-critical methods* has been the focus of past work [Hassanshahi et al. 2017; Jeong et al. 2017; Smaragdakis et al. 2014; Wei and Ryder 2015]. Those techniques are based on heuristics that seem to correlate with imprecision, but they do not provide a comprehensive understanding of how and where the imprecision is introduced in a context-insensitive pointer analysis. For example, introspective analysis [Smaragdakis et al. 2014] requires tuning multiple parameters involving sizes of various kinds of points-to sets, and data-driven analysis [Jeong et al. 2017] is parameterized by a collection of syntactic features and relies on machine learning for selecting good heuristics.

In this paper, we provide a more principled approach, named Zipper, to efficiently identify precision-critical methods, based on insights about how imprecision is introduced. The key observation is that most cases in which imprecision arises in a context-insensitive pointer analysis fit into three general patterns of *value flows*, which we call *direct*, *wrapped*, and *unwrapped* flows. Moreover, we show that these three kinds of value flows can be recognized efficiently. Based on information obtained from a fast, context-insentive pointer analysis, Zipper constructs a *precision flow graph* (PFG) that concisely models the relevant value flow. The identification of precision-critical methods can then be formulated as a graph reachability problem on the PFG and solved in negligible time, compared to the pointer analysis itself. By applying context sensitivity to the precision-critical methods identified by Zipper, a pointer analysis runs significantly faster than conventional techniques that apply context sensitivity indiscriminately to all methods, while retaining most of the precision.

In summary, we make the following contributions.

- We describe three fundamental patterns of value flows that help in explaining how and where most of the imprecision is introduced in a context-insensitive pointer analysis (Section 2).
- We present the Zipper approach to effectively recognize the three value-flow patterns and thereby identify the precision-critical methods that benefit from context sensitivity (Section 3). Zipper can guide context-sensitive pointer analysis to run faster while keeping most of its precision. In contrast to other techniques that apply context sensitivity selectively, the Zipper approach is based on a tangible understanding of imprecision and not on heuristics that require non-transparent machine learning or other tuning of analysis parameters.
- We provide an extensive experimental evaluation of our implementation of Zipper to evaluate its effectiveness (Section 4). On average, Zipper reports that only 38% of the methods are precision-critical, which preserves 98.8% of the precision (measured as average across a range of popular analysis clients) for a 2-object-sensitive pointer analysis with a context-sensitive heap, for a speedup of 3.4X and up to 9.2X. These results demonstrate that the three patterns of value flows indeed capture the vast majority of methods that benefit from context sensitivity.

2 CAUSES OF IMPRECISION IN CONTEXT-INSENSITIVE POINTER ANALYSIS

Our approach is based on the key insight that most of the precision loss in context-insensitive pointer analysis for Java can be expressed in terms of three basic patterns of value flows, or as combinations of these. We assume the reader is familiar with state-of-the-art context-sensitive pointer analysis techniques, e.g., as covered in several surveys [Ryder 2003; Smaragdakis and Balatsouras 2015; Sridharan et al. 2013], however, the precision loss patterns are independent of the chosen variant of context sensitivity, such as call-site sensitivity [Sharir and Pnueli 1981; Shivers 1991], object sensitivity [Milanova et al. 2005], and type sensitivity [Smaragdakis et al. 2011]. In this section, we introduce the three precision loss patterns and then describe three corresponding concrete examples (Sections 2.1–2.3). This characterization of precision loss provides the conceptual foundation for Zipper to identify precision-critical methods as explained in Section 3.

A context-insensitive analysis does not distinguish between different calls to a method but merges the incoming abstract values (or points-to sets, in the case of pointer analysis) [Sharir and Pnueli 1981]. Figure 1 shows a simple example. If method `m` is analyzed context-insensitively, then the two objects are mixed together, so the analysis conservatively concludes that both `x2` and `y2` may point to both the `A` object and the `B` object.

```
Object m(Object o){
  return o;
}
x1 = new A();
x2 = m(x1);
y1 = new B();
y2 = m(y1);
```

Fig. 1. Example of precision loss in context-insensitive analysis.

In contrast, a context-sensitive analysis would analyze `m` twice, corresponding to the two different call sites, and thereby conclude that `x2` can only point to an `A` object and `y2` can only point to a `B` object. The price of that extra precision is that the method needs to be analyzed multiple times, so context sensitivity should ideally only be applied when the precision gain outweighs the extra analysis time.

To characterize the relevant value flows, we first introduce some terminology.

Definition 2.1 (In and Out methods). Given a class C and a method M that is declared in C or inherited from C's super-classes, if M contains one or more parameters then *M is an In method of C*, and if M's return type is non-`void` then *M is an Out method of C*. (In the example in Figure 1, `m` is both an In and an Out method of the surrounding class.)

Definition 2.2 (Object wrapping and unwrapping). If an object O is stored in a field of an object W (or in an array entry of W, in case W is an array), then *O is wrapped into W*. Conversely, if an object O is loaded from a field of an object W (or from an array entry of W in case W is an array), then *O is unwrapped from W*. (The simple example in Figure 1 contains no wrapping or unwrapping.)

With these definitions in place, we can describe the three precision-loss patterns as different kinds of value flows, depicted in Figure 2.

Definition 2.3 (Direct flow). If, in some execution of the program, an object O is passed as a parameter to an In method M_1 of class C, and then flows (via a series of assignments, field

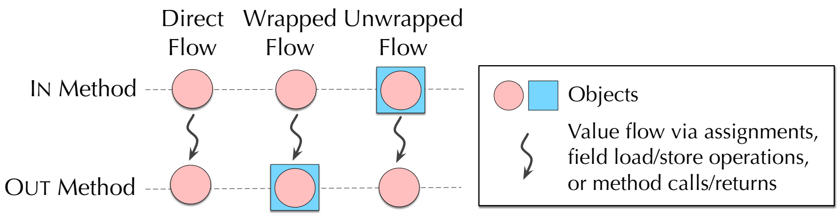
Direct Flow
Wrapped Flow
Unwrapped Flow
In Method
Out Method
Objects
Value flow via assignments, field load/store operations, or method calls/returns

Fig. 2. Three basic patterns of value flow that cause precision loss in context-insensitive analysis.

Direct Flow

```
1  class Person {                          15 }
2    String name; String id;               16 // Usage Code
3    void setName(String nm) {             17 void main() {
4      this.name = nm;                     18   Person p1 = new Person();
5      updateID();                         19   String name1 = new String("A");
6    }                                     20   p1.setName(name1);
7    void updateID() {                     21   String id1 = p1.getID();
8      String newName = this.name;         22
9      this.id = newName;                  23   Person p2 = new Person();
10   }                                     24   String name2 = new String("B");
11   String getID() {                      25   p2.setName(name2);
12     String id = this.id;                26   String id2 = p2.getID();
13     return id;                          27 }
14   }
```

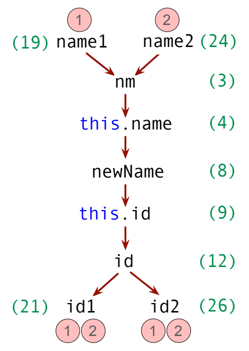

Fig. 3. Example of direct flow. (The line number for each variable/field reference on the right-hand side is shown in parentheses.)

load/store operations, method calls, or returns) to the return value of an Out method, M_2, of the same class C, then we say the program has *direct flow* from M_1 to M_2. (The example in Figure 1 is a simple instance of this pattern.)

Definition 2.4 (Wrapped flow). If, in some execution of the program, an object O is passed as a parameter to an In method M_1 of class C and then flows to a store operation that wraps O into an object W, where W subsequently flows to the result of an Out method, M_2, of the same class C, then we say the program has *wrapped flow* from M_1 to M_2. More generally, the wrapped flow pattern also covers value flow through multiple object wrapping steps, for example when W is itself wrapped into another object W', which flows to the return value of M_2.

Definition 2.5 (Unwrapped flow). If, in some execution of the program, an object O is passed as a parameter to an In method M_1 of class C and then flows to a load operation that unwraps an object U from O, where U subsequently flows to the return value of an Out method, M_2, of the same class C, then we say the program has *unwrapped flow* from M_1 to M_2. As in the previous definition, unwrapped flow also covers value flow through multiple object unwrapping steps.

2.1 Pattern 1: Direct Flow

The *setter and getter* example shown in Figure 3 demonstrates how direct flow is an indication of precision loss for a context-insensitive analysis. The `Person` class provides methods `setName` and `getID` to modify a person's name and retrieve his or her ID. Whenever a person's name is modified, the ID is updated accordingly (line 5).

After executing this code, `id1` in line 21 (resp. `id2` in line 26) points to object ① in line 19 (resp. ② in line 24) only. However, if the three methods of `Person` are analyzed using a context-insensitive pointer analysis, then `id1` and `id2` will both imprecisely point to objects ① and ②. Let us examine how this imprecision is connected to the direct flow pattern.

The right-hand side of Figure 3 illustrates how two objects ① and ②, respectively pointed to by `name1` and `name2`, first flow from their creation sites in lines 19 and 24 to the parameter `nm` of the In method `setName` in line 3, and then to `id` in line 12 through a series of store and load operations (line 4 $\longrightarrow$ line 8 $\longrightarrow$ line 9 $\longrightarrow$ line 12), and finally out of the Out method `getID` to `id1` and `id2` in lines 21 and 26. Hence, by Definition 2.3, the red arrows in Figure 3 form a direct flow.

Notice that with a context-insensitive analysis, objects ① and ② are merged in the same points-to set and further propagated according to this direct flow. In the analysis, the merged objects will

Wrapped Flow

```
1   class Collection {
2     Object elem;
3     void add(Object el) {
4       this.elem = el;
5     }
6     Iterator iterator(){
7       Object e = this.elem;
8       Iterator itr =
            new Iterator(e);
9       return itr;
10    }
11  }
12  class Iterator {
13    Object next;
14    Iterator(Object obj) {
15      this.next = obj;
16    }
17
18    Object next() {
19      return this.next;
20    }
21  }
22  //Usage Code
23  void main() {
24    Collection c1 = new Collection();
25    String s1 = new String("A");
26    c1.add(s1);
27    Iterator i1 = c1.iterator();
28    Object o1 = i1.next();
29
30    Collection c2 = new Collection();
31    String s2 = new String("B");
32    c2.add(s2);
33    Iterator i2 = c2.iterator();
34    Object o2 = i2.next();
35  }
```

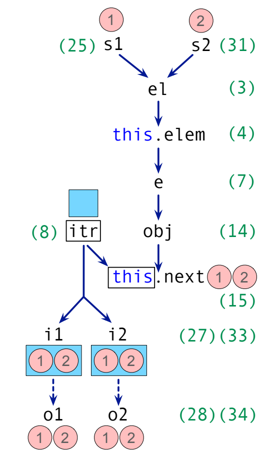

Fig. 4. Example of wrapped flow.

flow out of the Out method, causing id1 and id2 to point to spurious objects. Such imprecision will only get worse when some operations are further applied on id1 and id2 (not shown in this example), possibly polluting other parts of the program.

One way to avoid the imprecision is to apply context sensitivity to the methods that participate in the direct flow. We consider these to be *precision-critical methods*, since analyzing just one of them context-insensitively will likely introduce imprecision. With a context-sensitive analysis (for most variants of context sensitivity), in Figure 3, all variables and field references along the direct flow will be analyzed separately. For example, object sensitivity will use the two allocation sites at lines 18 and 23 as contexts. Accordingly, the merged paths along this direct flow are separated by the two contexts, like unzipping a zipper—hence the name of our technique. A similar strategy of separating merged paths also applies to wrapped and unwrapped flows, as shown next.

2.2 Pattern 2: Wrapped Flow

The *collection and iterator* example shown in Figure 4 demonstrates how the wrapped flow pattern yields precision loss for a context-insensitive analysis. To keep the example simple, the collection only stores one element, however the code pattern is directly analogous to realistic code, for arbitrarily-sized collections. Class Collection provides an add method to add an element to the collection and an iterator method to return an iterator that has a pointer, next, pointing to the collection element (as set in line 15). The element is passed as an argument to the newly created iterator (line 8), which establishes a connection between the collection and its iterator. Two objects ① (line 25) and ② (line 31) are stored in two different collections, c1 (line 26) and c2 (line 32). The two objects are then accessed by the iterators of the collections (lines 28 and 34).

After executing the code, o1 in line 28 (resp. o2 in line 34) points to object ① (resp. ②) only. However, if *any of the four* methods of Collection and Iterator are analyzed context-insensitively, o1 and o2 will both imprecisely point to both objects ① and ②. Let us examine how this imprecision is connected to the wrapped flow pattern.

As shown on the right-hand side of Figure 4, similarly to the direct flow example in Figure 3, objects ① and ② flow into the In method add of class Collection, and then further to lines 7, 8, and 14. Unlike a direct flow, the objects ① and ② do not directly flow out of the Out method `iterator` of class `Collection`; instead, a wrapper `Iterator` object, $\square$, (created on line 8) in which object ① or ② is stored, flows out of this Out method.

Object wrapping (Definition 2.2) occurs in line 15: objects ① and ② (pointed to by `obj`) are stored into the `next` field of the object pointed to by `this`, and `this` points to the receiver object of the constructor call in line 8, which is also pointed to by `itr` in line 8. As a wrapper object (that stores object ① or ②) flows out of an Out method of the same class, by Definition 2.4, the solid blue arrows in Figure 4 form a wrapped flow.

With a context-insensitive analysis, objects ① and ② are merged in the same points-to set and further propagated according to this wrapped flow. However, unlike a direct flow, imprecision is not introduced until the access operation (e.g., the `next` calls in lines 28 and 34) is applied on the flowing-out wrapper object, causing variables `o1` and `o2` to point to spurious objects. The wrapper objects carry the flowing-in objects, which originate from outside the class, so context sensitivity can separate the merged objects all along their flow through the `Collection` class.

The example also helps illustrate some subtleties of the flow definitions. Note that the precision loss patterns are expressed relative to a class: for each of the three patterns, the In method and the Out method must be *in the same class*, although the value flow may involve other classes, as described in Definitions 2.3—2.5. Intuitively, if the precision loss flows introduced in *each class* (through method calls on the objects of the class) could be identified and then avoided by use of context sensitivity, the imprecision of the *whole program* could be accordingly controlled via such a divide-and-conquer scheme. In addition, this design choice enables an efficient and elegant algorithm for identifying occurrences of the patterns in a given program, by considering each class one by one, as explained in Section 3.

Therefore, the dashed arrows (bottom right of Figure 4) formed by calling the `next` method in lines 28 and 34, do not belong to the wrapped flow, because the calls happen after the wrapper objects flow out from the Out method of class `Collection`. Thus, as explained in Section 2.1, only methods `add` and `iterator` (in `Collection`) and the constructor `Iterator` (in `Iterator`) are included in the wrapped flow and thus considered precision-critical. However, if we consider In and Out methods from the point of view of class `Iterator`, then method `next` is also precision-critical, since it is involved in a direct flow together with the `Iterator` constructor, much like the *setter* and *getter* methods in Section 2.1.

2.3 Pattern 3: Unwrapped Flow

We use a *synchronized box* example (based on classes `SynchronizedSet` and `Set` in the JDK but heavily simplified) to illustrate an unwrapped flow, as shown in Figure 5. Class `SyncBox` encapsulates class `Box` by providing synchronization in the encapsulating method `getItem` (lines 6–12). Two objects ① and ② are stored into two `Box` objects (represented by $\square$ and pointed to by `b1` and `b2` in lines 27 and 32), which are further stored into two `SyncBox` objects (lines 28 and 33).

After executing the code, `o1` in line 29 (resp. `o2` in line 34) points to object ① (resp. ②) only. However, if any of the four methods of classes `SyncBox` and `Box` are analyzed context-insensitively, `o1` and `o2` will both imprecisely point to both objects ① and ②. Let us examine how this imprecision is connected to the unwrapped flow pattern.

As shown on the right-hand side of Figure 5, similar to the direct flow in Figure 3, two `Box` objects $\boxed{1}$ and $\boxed{2}$ (pointed to by `b1` and `b2`, respectively) flow into the body of class `SyncBox` through its constructor, which acts as an In method, and then further to `b` in line 8. Unlike in a direct flow, the flowing-in objects $\boxed{1}$ and $\boxed{2}$ do not flow out of the Out method `getItem` of class `SyncBox`; instead, the two unwrapped objects ① and ② (respectively stored in $\boxed{1}$ and $\boxed{2}$) are the ones that flow out of this Out method.

Unwrapped Flow

```
1  class SyncBox {
2    Object box;
3    SyncBox(Box box) {
4      this.box = box;
5    }
6    Object getItem() {
7      synchronized(this) {
8        Box b = this.box;
9        Object o = b.getItem();
10       return o;
11     }
12   }
13 }
14 class Box {
15   Object item;
16   Box(Object item) {
17     this.item = item;
18   }
19   Object getItem() {
20     Object it = this.item;
21     return it;
22   }
23 }
24 //Usage Code
25 void main() {
26   String s1 = new String("A");
27   Box b1 = new Box(s1);
28   SyncBox sb1 = new SyncBox(b1);
29   Object o1 = sb1.getItem();
30
31   String s2 = new String("B");
32   Box b2 = new Box(s2);
33   SyncBox sb2 = new SyncBox(b2);
34   Object o2 = sb2.getItem();
35 }
```

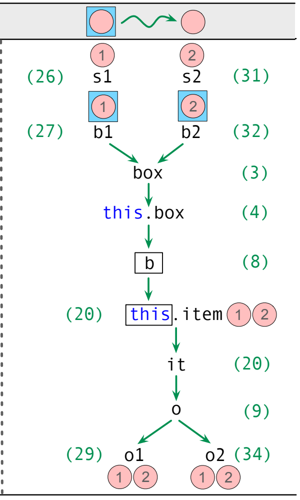

Fig. 5. Example of unwrapped flow.

Object unwrapping (Definition 2.2) occurs in line 20, as a result of the call in line 9: the Box objects ($\boxed{1}$ and $\boxed{2}$ pointed to by b) are the receiver objects of this virtual call, and this in line 20 will also point to them during pointer analysis. The load operation in line 20 lets the unwrapped objects (① and ②) flow to it (line 20), and finally to o1 and o2 (lines 29 and 34) through consecutive method return values (line 21 ⟶ line 9 and then line 10 ⟶ lines 29 and 34). As the unwrapped objects (retrieved from the flowing-in objects) flow out of an Out method of the same class, by Definition 2.5, the green arrows (in Figure 5) form an unwrapped flow.

We can observe that objects $\boxed{1}$ and $\boxed{2}$ (and hence the unwrapped objects ① and ② they contain) are merged in the same points-to set and further propagated according to this unwrapped flow. Although the flowing-in objects do not flow out of an Out method of the same class to introduce imprecision, the unwrapped objects do, causing the receiving variables, in this case o1 and o2 (lines 29 and 34), to point to spurious objects.

Note that the program points where the unwrapped objects are stored in the flowing-in objects (lines 26–27 and 31–32) do not belong in the unwrapped flow, as the objects have not yet entered the In method of class SyncBox. Thus, only constructor SyncBox, method getItem (in SyncBox), and method getItem (in Box) belong in the unwrapped flow and are considered precision-critical. However, as in the explanation of the wrapped flow example in Section 2.2, if we consider In and Out methods from the point of view of class Box, its constructor, Box, will still be analyzed context-sensitively as it is part of a direct flow (together with the getItem method in Box).

Finally, some imprecision cannot be described by one pattern alone but only by combinations. Consider the example of an object W that flows into an In method, where an object O is unwrapped from W. Then O is wrapped into another wrapper object, W', which flows out from an Out method of the same class. Imprecision may arise in this case, and although none of the three basic flow patterns in isolation match this flow, it is captured by a combination of unwrapped and wrapped flows. Zipper identifies not only occurrences of the three patterns but also such combinations. Our experiments (Section 4) show that the patterns and their combinations account for essentially all the imprecision that may appear in context-insensitive analysis.

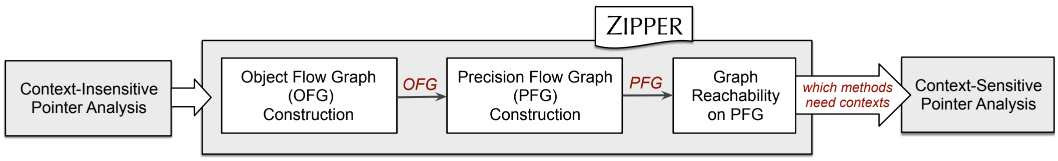
ZIPPER
Context-Insensitive Pointer Analysis
Object Flow Graph (OFG) Construction
OFG
Precision Flow Graph (PFG) Construction
PFG
Graph Reachability on PFG
which methods need contexts
Context-Sensitive Pointer Analysis

Fig. 6. Overview of ZIPPER.

3 ZIPPER

This section introduces ZIPPER: our approach to identifying precision-critical methods based on the precision loss patterns of Section 2. Even if the patterns successfully characterize the main causes of precision loss in context-insensitive analysis, two challenges remain. First, the precision loss patterns are defined in *dynamic* execution terms, while ZIPPER has to capture the potential for these patterns using *static* information. Second, useful static information has to be computable from a mere context-insensitive analysis, in order to guide a context-sensitive one. That is, the potential for precision loss has to be detected from an analysis that already exhibits this loss. The ZIPPER approach is defined with these goals in mind, and manages to make context-sensitive pointer analysis run faster while preserving most of its precision.

We present the overview of ZIPPER in Section 3.1 and the concepts of *object flow graphs* and *precision flow graphs* in Sections 3.2 and 3.3, respectively.

3.1 Overview of ZIPPER

The goal of ZIPPER is to efficiently recognize the precision-critical methods in a given program. The central part of ZIPPER is the notion of *precision flow graphs* (PFGs) that allow us to express all three precision loss patterns in a uniform way, in the sense that each kind of flow can be represented by a path in a PFG. Intuitively, a PFG is much like the right-hand side graphs of Figures 3–5, but replacing the field expressions by the abstract objects and their fields. Via the PFGs, we can convert the problem of identifying precision-critical methods to an abstract graph computation. All methods that are involved in one of the three kinds of flows can be efficiently extracted by solving a simple graph reachability problem on the PFGs.

Constructing the PFGs requires information about how objects flow in the program. We leverage the concept of *object flow graphs* (OFGs) [Tonella and Potrich 2005] as explained in Section 3.2. The OFG for a program allows tracing the flow of objects through local assignments, calls and returns, and field load and store operations in the program. Therefore, it can naturally express the direct flow pattern, in a static analysis that approximates the dynamic flows of objects. However, the original OFG formulation does not represent wrapped and unwrapped flows, thus we cannot directly use it to identify precision-critical methods. For this reason, we build the PFGs on top of the OFG to uniformly express all three precision loss patterns.

Figure 6 shows the overall structure of ZIPPER, which itself contains three components: the *object flow graph construction*, the *precision flow graph construction*, and the *graph reachability computation*. First, a fast but imprecise context-insensitive pointer analysis is performed as a pre-analysis for ZIPPER. To simplify the discussion, we assume that the pre-analysis abstracts objects by their allocation-sites [Chase et al. 1990], but our technique also works for other object abstractions [Kanvar and Khedker 2016]. This pre-analysis provides the information for the OFG construction, in the form of a relation $pt(\mathsf{v})$ that captures the points-to set for each variable v. Based on the OFG, a PFG is constructed *for each class*. Afterwards, ZIPPER computes graph reachability on each PFG to determine which methods are precision-critical. Finally, a selective context-sensitive

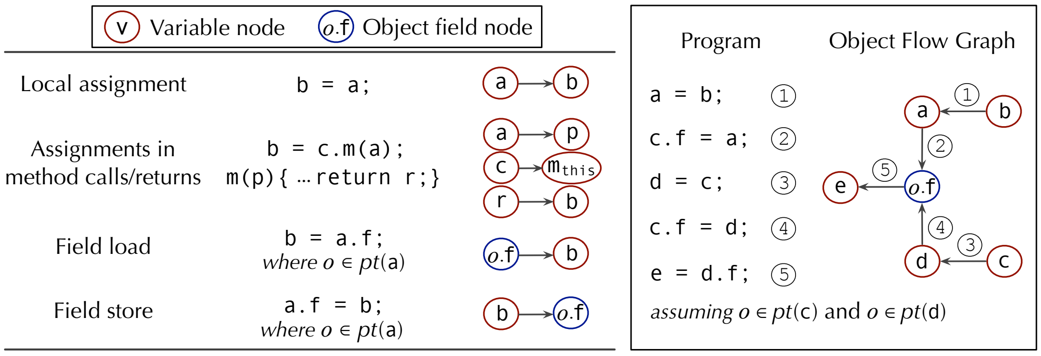

Fig. 7. Object flow graph construction, with an example.

pointer analysis is performed, guided by Zipper's results, so that the pointer analysis applies context sensitivity to only the precision-critical methods reported by Zipper.

3.2 Object Flow Graphs

The *object flow graph* (OFG) of a program, as in its original form by Tonella and Potrich [2005], is a directed graph that expresses how objects flow in the program. The nodes in the OFG represent program pointers, which can point to objects, and the edges represent basic object flow among the pointers. More precisely, the OFG contains a node for each variable in the program and for each field of each abstract object. Objects are abstracted in the same way as in the pre-analysis, as described in Section 3.1: we here assume allocation-site abstraction is being used, which is the most common choice, but the technique also works for other choices. An edge a⟶b in the OFG means that the objects pointed by pointer a may flow to (and also be pointed to by) pointer b. Another way to view the OFG is that it is the subset constraint graph in an Andersen-style points-to analysis [Andersen 1994; Sridharan et al. 2013].

Tonella and Potrich [2005] propose to build the OFG with more precision by cloning the variables of a method for each of its receiver objects (conceptually like object sensitivity [Milanova et al. 2002, 2005]), so that the flow involved in different receiver objects of the same method can be distinguished. However, this is unnecessary for Zipper, since it builds the OFG based on the results of a context-insensitive analysis, and all flow queries are done at the class level instead of the object level, as explained in Section 2. Therefore, we perform no such cloning.

Due to the close connection between OFGs and Andersen-style analysis, constructing the OFG is trivial, based on the points-to relation *pt*(v) provided by the context-insensitive pre-analysis. Figure 7 illustrates this construction. The left-hand side of Figure 7 lists (from left to right) the four basic object flows, the related Java statements that induce the flows, and the corresponding graph edges in the OFG.

Consider the code fragment and its corresponding OFG on the right-hand side of Figure 7. There are five statements labeled ① – ⑤, and each statement causes an edge (with the same label) to be added to the OFG. With the OFG, the object flow information can be directly obtained simply by checking graph reachability without the need to explicitly track alias information among variables or field accesses. For example, variable e is reachable from b in the OFG, which means that the objects pointed to by b may flow to (and also be pointed to by) e.

As a result, direct flows can be expressed naturally by the paths in the OFG, however, that is not the case for wrapped and unwrapped flows. In the next section, we describe how to augment the OFG to express all three kinds of flows.

Algorithm 1: PfgBuilder

```
Input  : OFG    (Object Flow Graph)
         $c$    (Input class)
         S      (Set of statements in the program)
Output : PFG$_c$  (Precision Flow Graph for class $c$)
1  PFG$_c$ ← { }, $VisitedNodes$ ← { }, $WUEdges$ ← { }
2  foreach $m \in$ In$_c$ do
3      foreach parameter $p$ of $m$ do
4          Dfs($N_p$) where $N_p$ is the OFG node for $p$
5  return PFG$_c$
6  Function Dfs($N$)
7      if $N \in VisitedNodes$ then
8          return
9      add $N$ to $VisitedNodes$
10     if $N$ is a variable node $N_{\mathsf{a}}$ then
11         foreach $\boxed{\texttt{b = a.f}}$ ∈ S do                // Handling unwrapped flow
12             add $N_{\mathsf{a}} \to N_{\mathsf{b}}$ to $WUEdges$
13         foreach $\boxed{\texttt{b.f = a}}$ ∈ S do                  // Handling wrapped flow
14             foreach o ∈ $pt$(b) do
15                 add $N_{\mathsf{a}} \to N_{[\mathsf{o}]}$ to $WUEdges$
16     foreach $N \to N'$ ∈ OFG ∪ $WUEdges$ do
17         add $N \to N'$ to PFG$_c$
18         Dfs($N'$)
```

3.3 Precision Flow Graphs and Graph Reachability

We first explain how to construct *precision flow graphs* (PFGs) and then how to identify precision-critical methods by performing graph reachability on each PFG.

Precision Flow Graph Construction. As explained in Section 3.2, one OFG is built for the entire program. Since the PFGs serve to express the three kinds of precision loss patterns, which are all defined relative to a class, as explained in Section 2, we construct one PFG for each class in the program. As the OFG can already describe direct flow (Section 3.2), the task of building the PFG is to add edges that can express the other two kinds of flows: wrapped and unwrapped flows. Algorithm 1 (PfgBuilder) shows how to build PFG$_c$ for a given class c. For simplicity, we represent the PFG and the OFG by their sets of graph edges, and the graph nodes are implicitly those that appear in the edge sets.

Three sets are initialized to empty sets in line 1: the PFG edges, the set of visited nodes, and *WUEdges*, which denotes a set of extra edges for wrapped and unwrapped flows. As all three kinds of flows begin from the parameters of an In method (see Section 2), the algorithm starts by iterating through those methods (lines 2–3, where In$_c$ denotes the set of In methods of the input class c).

Function Dfs (line 6) traverses the input OFG and adds the edges for wrapped and unwrapped flows. As a result, the returned PFG$_c$ (line 5) includes all the nodes that can be reached from each parameter of In methods of c, through direct, wrapped, and unwrapped flows, or combinations of these. Specifically, unwrapped and wrapped flows are handled in lines 11–12 and lines 13–15, respectively, by adding the corresponding edges to *WUEdges*. Finally, the generated PFG includes

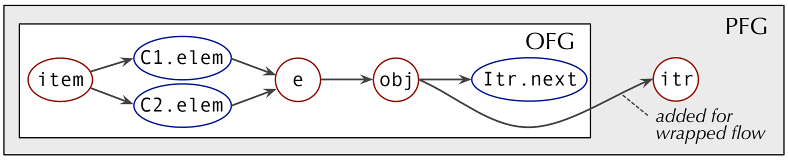

Fig. 8. A partial PFG for class `Collection` in Figure 4 (wrapped flow). `C1`, `C2`, and `Itr` denote the objects of classes `Collection` and `Iterator` allocated in lines 24, 30, and 8 in Figure 4, respectively.

direct flows (from the OFG) and wrapped/unwrapped flows (from *WUEdges*) via the statements in lines 16–17. Now let us see the details of handling wrapped and unwrapped flows.

Recall that each OFG node represents either a variable or a field of an abstract object. If node N in line 10 is a variable node $N_{\mathtt{a}}$, then for every load operation (`b = a.f` in line 11) that may load the (unwrapped) objects (which are stored in a field of an object pointed to by `a`) to variable `b`, we add an edge from node $N_{\mathtt{a}}$ to node $N_{\mathtt{b}}$. This allows us to model unwrapped flow, as defined in Definition 2.5 and illustrated in Section 2.3.

The most intricate part of the algorithm is lines 13–15, which handle wrapped flows. If node N in line 10 is a variable node $N_{\mathtt{a}}$, then for every store operation (`b.f = a` in line 13) that can store the objects (pointed to by `a`) in wrapper objects `o` pointed to by `b` (line 14), we add an edge from node $N_{\mathtt{a}}$ to $N_{[\mathtt{o}]}$. Here we use the notation `[o]` to denote the variable that the abstract object `o` was originally assigned to when created: for example, if `o` is created at a statement `v = new ...` then `[o]` is the variable `v`. These added edges enable tracking wrapped flow as defined in Definition 2.4 and illustrated in Section 2.2. As an example, for the object wrapping `this.next = obj` of line 15 in Figure 4, *pt*(`this`) contains an abstract object created at `itr = new Iterator(e)` in line 8, so we add an edge from `obj` to `itr`.

Note that if (in line 15) instead of adding an edge from $N_{\mathtt{a}}$ to $N_{[\mathtt{o}]}$ we had added an edge from $N_{\mathtt{a}}$ to $N_{\mathtt{b}}$ (mirroring the handling of unwrapped flows), we would miss some flows. Conceptually, according to Definition 2.4, modeling wrapped flow requires tracking the wrapper objects (from where they are created) rather than the variable `b` in the store operation `b.f = a` (line 13). For example, in the case of Figure 4, consider the store operation `this.next = obj` (line 15) where `this` (line 15) and `itr` (line 8) both point to the `Iterator` object created in line 8. If we added an edge from node $N_{\mathtt{obj}}$ to node $N_{\mathtt{this}}$ (rather than $N_{\mathtt{obj}}$ to $N_{\mathtt{itr}}$), the flow tracking from $N_{\mathtt{this}}$ would not lead to the `return` statement (line 9) in the Out method, because the wrapped flow flows out through node $N_{\mathtt{itr}}$ in this case. However, it is safe to add an edge to node $N_{\mathtt{itr}}$ instead (as we do in line 15 in Algorithm 1) since the wrapper object is originally assigned to `itr`, so that the flow of the wrapper object is taken into account as required by Definition 2.4.

Through Algorithm 1, we can see that wrapped and unwrapped flows can be naturally expressed in the PFG by handling the store/load operations (lines 10–15) recursively during the graph traversal. In addition, the newly added edges for wrapped and unwrapped flows build new connections with existing OFG edges that model direct flows. As a result, the generated PFG also naturally expresses combinations of all three kinds of flows.

Figure 8 shows a partial PFG example for class `Collection` from Figure 4. The existing OFG is constructed following the rules in Figure 7. In Figure 8, in the three object field nodes (`C1.elem`, `C2.elem`, and `Itr.next`), the abstract objects respectively denoted by `C1`, `C2`, and `Itr` represent the objects of classes `Collection` and `Iterator`. Node `obj` corresponds to $N_{\mathtt{a}}$ in line 15 in Algorithm 1; the edge from node `obj` to node `Itr.next` corresponds to the store operation `this.next = obj` in line 15 in Figure 4, and also the store operation `b.f = a` in line 13 in Algorithm 1. According to

Algorithm 2: PcmCollector

```
Input  :  $c$          (Input class)
          $PFG_c$      (Precision Flow Graph for class $c$)
Output :  $PCM_c$      (Precision-Critical Methods for class $c$)
1  $FlowNodes \leftarrow \{\}$, $PCM_c \leftarrow \{\}$
2  foreach $m \in$ Out$_c$ do
3      foreach return variable $r$ of $m$ do
4          $FlowNodes \bigcup=$ NodesCanReach($N_r$, $PFG_c$)        // Backward graph reachability
5  foreach $N \in FlowNodes$ do
6      if $N$ is a variable node $N_a$ and $a$ is declared in $m$ then
7          add $m$ to $PCM_c$
8      if $N$ is an object field node $N_{o.f}$ and $o$ is allocated in $m$ then
9          add $m$ to $PCM_c$
10 return $PCM_c$
```

line 15 in Algorithm 1, an edge that enables tracking the wrapped flow is added in Figure 8 from node `obj` to node `itr`, since `[Itr]` is the variable `itr`.

Graph Reachability on Precision Flow Graphs. We now explain how Zipper extracts the precision-critical methods based on the PFGs. Generally, Zipper first computes all the nodes that are involved in the three kinds of flows by solving a simple graph reachability problem on the PFG, and then collects the methods that contain the nodes as the precision-critical methods.

Given a class c, each flow in the precision loss patterns corresponds to a path from a parameter node of an In method of c to a return variable node of an Out method of c in PFG_c. Therefore, obtaining the statements that are involved in the flows is equivalent to computing which nodes are reachable from a parameter of an In method and can also reach a return variable of an Out method in PFG_c. Since Zipper builds PFG_c starting from the parameters of the In methods (lines 2–3 in Algorithm 1), all nodes in PFG_c are reachable from the In methods. Therefore, we only need to find out which nodes in PFG_c can reach the return variables of Out methods of class c.

Algorithm 2 (PcmCollector) defines the collection of precision-critical methods for an input class c based on PFG_c. In line 1, two sets are initialized to empty: *FlowNodes* denotes the set of nodes that are involved in the flows from In methods to Out methods of class c, and PCM_c denotes the set of precision-critical methods for class c, i.e., the methods that contain the nodes in *FlowNodes*.

In lines 2–4, PcmCollector fills *FlowNodes* by iterating through the return variables of all Out methods of c (denoted by Out$_c$) and collecting all nodes that can reach the return variables in PFG_c. The function NodesCanReach used in line 4 is a standard backward graph reachability algorithm which traverses the PFG_c starting from N_r and returns all nodes that can reach N_r on PFG_c.

In lines 5–9, PcmCollector fills PCM_c. There are two kinds of nodes in PFG_c that are handled differently. For a variable node N_a, PcmCollector adds the method where the variable a is declared to PCM_c (lines 6–7). For an object field node $N_{o.f}$, PcmCollector adds the method where the abstract object o is allocated to PCM_c (lines 8–9).

As a result, the algorithm collects the precision-critical methods for each class in a given program. With this information, Zipper can guide context-sensitive pointer analyses to apply context sensitivity only for the precision-critical methods.

The precise statements of Algorithms 1 and 2 capture the design choices of Zipper. Inferences on flow patterns are made on a per-class basis, and context sensitivity is applied on a per-method basis. It is easy to imagine applying context sensitivity at a finer granularity. That is, we could apply context sensitivity to only the variables and object fields that are involved in the flows in the precision loss patterns (i.e., the nodes stored in *FlowNodes* in Algorithm 2) instead of the entire containing methods. In this way, although within the same precision-critical methods, other variables and object fields that are irrelevant to precision loss patterns can be analyzed context-insensitively, which may lead to better efficiency. For simplicity, in this paper we only consider context sensitivity at the granularity of methods, and leave the potential of more refined options for future work.

4 EVALUATION

In this section, we investigate the following research questions for evaluation.

RQ1. Is ZIPPER-guided pointer analysis precise and efficient?
- (a) How much of the precision of a conventional analysis can ZIPPER preserve?
- (b) How fast is ZIPPER-guided pointer analysis compared to a conventional analysis?
- (c) What is the overhead of running ZIPPER?

(By "conventional", we mean a context-sensitive pointer analysis that applies context sensitivity to all methods.)

RQ2. How does ZIPPER-guided pointer analysis compare to state-of-the-art alternative techniques (specifically, introspective analyses [Smaragdakis et al. 2014]) that also apply context sensitivity for only a subset of the methods, in terms of precision and efficiency?

RQ3. What is the effect of each of ZIPPER's precision loss patterns on the analysis results?
- (a) How many methods does ZIPPER consider precision-critical, and how does each precision loss pattern contribute to this number?
- (b) How does each of the precision loss patterns affect the precision and efficiency of ZIPPER-guided pointer analysis?

Implementation. We have implemented ZIPPER as an open-source stand-alone tool in Java, available at http://www.brics.dk/zipper. Benefiting from simple insights and algorithms, ZIPPER's core implementation contains less than 1500 lines of Java code. In addition, ZIPPER is designed to work with various pointer analysis frameworks, such as DOOP [Bravenboer and Smaragdakis 2009], WALA [WALA 2018], CHORD [Naik et al. 2006], and SOOT [Vallée-Rai et al. 1999]. To investigate its effectiveness, we have integrated ZIPPER with DOOP, a state-of-the-art whole-program pointer analysis framework for Java. Interacting with existing context-sensitive pointer analysis is simple, as ZIPPER's output is just a set of precision-critical methods, as shown in Figure 6. For example, we only need to slightly modify three Datalog rules in DOOP to enable DOOP to apply context sensitivity to only the precision-critical methods reported by ZIPPER. We expect a similarly simple integration for other pointer analysis tools.

Experimental Settings. We run all experiments on a machine with an Intel Xeon (E5) 2.6GHz CPU and 48G memory. The time budget is set to 1.5 hours as in previous work [Jeong et al. 2017; Kastrinis and Smaragdakis 2013; Smaragdakis et al. 2014]. We evaluate ZIPPER using a large OpenJDK (1.6.0_24) library and 10 large Java programs: five are popular real-world applications (the first five entries in Table 1) and five are from the standard DaCapo 2006 benchmarks [Blackburn et al. 2006] (the last five entries in Table 1). We discuss the reason for this subset of the DaCapo benchmarks after introducing the metrics and analysis settings.

In RQ1, we consider a 2-object-sensitive pointer (`2obj`) analysis (with one context element for heap objects) [Milanova et al. 2002, 2005] as the conventional context-sensitive pointer analysis we seek to match in terms of precision. `2obj` is regarded as the most practical high-precision pointer analysis for Java [Lhoták and Hendren 2006; Smaragdakis et al. 2011; Tan et al. 2016] and is widely adopted in recent literature [Hassanshahi et al. 2017; Jeong et al. 2017; Kastrinis and Smaragdakis 2013; Scholz et al. 2016; Smaragdakis et al. 2013, 2014; Tan et al. 2017; Thiessen and Lhoták 2017] and analysis tools, including popular static analysis frameworks for Android [Arzt et al. 2014; Gordon et al. 2015]. Relative to other *k*-object-sensitive analyses, `2obj` is significantly more precise than `1obj` [Kastrinis and Smaragdakis 2013; Smaragdakis et al. 2011], and `3obj` does not scale for most DaCapo benchmarks [Tan et al. 2017].

In RQ2, we compare Zipper with the introspective analysis of Smaragdakis et al. [2014], which is the most closely related state-of-the-art analysis that employs context sensitivity only for a subset of the methods. These methods are selected by a pre-analysis according to two heuristics (the pre-analysis is also based on a fast context-insensitive pointer analysis, like Zipper), resulting in two variants of introspective analyses, IntroA and IntroB. (The naming and heuristics are from Smaragdakis et al. [2014]. The Doop integration of Zipper is using the version published for the artifact evaluation process of PLDI'14, which contains the exact setup for these algorithms, for direct comparison.) Generally, IntroA is faster but less precise than IntroB.

In the DaCapo benchmarks, `2obj` fails to scale for `jython` and `hsqldb` within 1.5 hours. Zipper also cannot help scale for these two known problematic benchmarks [Kastrinis and Smaragdakis 2013; Smaragdakis et al. 2011; Tan et al. 2016, 2017], as, unlike the introspective analysis of Smaragdakis et al. [2014], Zipper is designed to keep most of the analysis precision: its *precision-guided principle* prevents it from further removing more contexts, since that could degrade precision. Regarding introspective analysis, IntroB also fails to scale for `jython` but scales for `hsqldb`; IntroA scales for both but only achieves precision slightly better than a context-insensitive analysis. Consequently, to provide an observable precision baseline (i.e., the most precise results achieved by `2obj`), we consider the remaining five large DaCapo benchmarks for which `2obj` is scalable. We will examine how Zipper performs on the smaller, trivially-scalable benchmarks in Section 4.4.

4.1 RQ1: Precision and efficiency of Zipper-guided pointer analysis

In this section, we first examine the precision and efficiency of Zipper-guided pointer analysis by comparing it with `2obj` as explained above, and then show the overhead of running Zipper itself. As a conventional context-sensitive pointer analysis, to produce high precision, `2obj` applies context sensitivity to each method of the program indiscriminately. This is still the mainstream context-sensitivity scheme deployed in most pointer analysis frameworks for Java [Bravenboer and Smaragdakis 2009; Naik et al. 2006; WALA 2018]) and Android [Arzt et al. 2014; Gordon et al. 2015].

Table 1 shows the results of all analyses. Each program has five rows of data, respectively representing context-insensitive pointer analysis (ci), conventional object-sensitive pointer analysis (2obj), Zipper (zipper-2obj), and two introspective pointer analyses (introA-2obj and introB-2obj). The last two analyses will be discussed in Section 4.2.

4.1.1 How much precision of a conventional analysis is preserved by Zipper. To measure precision, we consider four independently useful client analyses, (subsets of which) also used as the precision metrics in past literature [Jeong et al. 2017; Kastrinis and Smaragdakis 2013; Lhoták and Hendren 2006; Smaragdakis et al. 2014; Sridharan and Bodík 2006; Tan et al. 2017]: a cast-resolution analysis (metric: the number of cast operations that may fail, denoted #fail-cast), a devirtualization analysis (metric: the number of virtual call sites that cannot be disambiguated into monomorphic calls, denoted #poly-call), a method reachability analysis (metric: the number of reachable methods, denoted #reach-mtd), and a call-graph construction analysis (metric: the number of call graph edges, denoted #call-edge). These metrics should give a thorough idea of analysis precision for useful clients. The results are shown in the last four columns in Table 1. In all cases, lower is better.

Table 1. Performance and precision metrics for context-insensitive (ci), conventional object-sensitive (2obj), Zipper-guided (zipper-2obj), and introspective object-sensitive (introX-2obj) pointer analyses.

Program	Pointer analysis	Time (s)	#fail-cast	#poly-call	#reach-mtd	#call-edge
batik	ci	82	2 961	4 681	19 197	101 616
	2obj	3 137	1 606	3 491	16 859	76 807
	zipper-2obj	927	1 614	3 501	16 863	76 858
	introA-2obj	232	2 675	4 262	19 011	97 120
	introB-2obj	2 146	2 149	3 997	18 703	90 126
checkstyle	ci	50	1 114	1 444	9 866	57 490
	2obj	1 912	581	1 035	9 513	48 809
	zipper-2obj	355	607	1 059	9 526	48 945
	introA-2obj	124	970	1 206	9 769	55 736
	introB-2obj	1 566	792	1 134	9 595	51 437
sunflow	ci	61	3 003	4 113	19 773	106 410
	2obj	1 124	1 837	3 385	19 245	89 866
	zipper-2obj	520	1 869	3 391	19 247	89 902
	introA-2obj	153	2 764	3 796	19 651	103 536
	introB-2obj	404	2 346	3 529	19 429	95 602
findbugs	ci	52	2 508	2 925	13 036	77 370
	2obj	2 321	1 409	2 182	12 657	65 836
	zipper-2obj	830	1 437	2 190	12 662	65 880
	introA-2obj	191	2 271	2 422	12 960	73 681
	introB-2obj	422	2 024	2 372	12 882	70 725
jpc	ci	58	2 370	5 013	17 146	96 669
	2obj	515	1 392	4 222	15 852	81 030
	zipper-2obj	211	1 415	4 231	15 857	81 072
	introA-2obj	130	2 169	4 703	17 038	95 170
	introB-2obj	331	1 736	4 327	16 001	85 316
eclipse	ci	23	1 139	1 334	8 465	45 474
	2obj	126	546	980	7 911	38 151
	zipper-2obj	66	586	1 013	7 927	38 369
	introA-2obj	58	977	1 118	8 319	43 781
	introB-2obj	72	764	1 046	8 001	39 876
chart	ci	46	1 810	1 852	12 064	63 453
	2obj	244	883	1 378	11 330	52 374
	zipper-2obj	77	910	1 384	11 334	52 399
	introA-2obj	126	1 580	1 613	11 952	61 323
	introB-2obj	183	1 236	1 497	11 518	55 594
fop	ci	74	2 458	3 585	17 154	84 330
	2obj	1 022	1 446	2 844	16 438	71 408
	zipper-2obj	457	1 471	2 860	16 442	71 478
	introA-2obj	197	2 206	3 246	17 007	82 113
	introB-2obj	512	1 804	2 979	16 571	75 770
xalan	ci	39	1 182	1 898	9 705	51 302
	2obj	985	533	1 522	9 047	44 871
	zipper-2obj	107	568	1 542	9 129	45 332
	introA-2obj	111	1 129	1 765	9 637	50 659
	introB-2obj	705	723	1 579	9 119	45 904
bloat	ci	31	1 924	2 014	8 939	61 150
	2obj	3 128	1 193	1 427	8 470	53 143
	zipper-2obj	2 704	1 224	1 449	8 486	53 289
	introA-2obj	57	1 809	1 690	8 869	60 111
	introB-2obj	135	1 621	1 522	8 626	55 455
	zipper-2obj*	52	1 310	1 511	8 538	54 049

```
1  class BufferedReader{
2    Reader in;
3    BufferedReader(Reader in){
4      this.in = in;
5    }
6    void close(){in.close();}
7  }
8  //Usage Code
9  InputStreamReader isReader = new InputStreamReader();
10 BufferedReader reader1 = new BufferedReader(isReader);
11 reader1.close();
12 FileReader fReader = new FileReader();
13 BufferedReader reader2 = new BufferedReader(fReader);
14 //reader2.close();
```

Fig. 9. Example of the no-out flow case.

Comparing Zipper with the conventional pointer analysis 2obj, we see that Zipper is able to achieve nearly identical precision as 2obj for every metric in every program. In summary, on average, 98.8% of the precision of 2obj can be preserved considering all client analyses.[1] Specifically, the average number for each client analysis is 96.8% for #fail-cast, 98.9% for #poly-call, 99.8% for #reach-mtd and 99.7% for #call-edge.

Zipper can produce such great precision because it is designed according to its precision-guided principle: all the methods that are involved in the three basic flows (direct, wrapped, and unwrapped flows), or their combinations, will be analyzed context-sensitively. Since the three flows capture the essence of value flows in Java programs where imprecision may arise through method calls (as explained in Section 2), most context-related imprecision can be discovered by Zipper. However, on average, Zipper still misses 1.2% of the precision. Although these cases are rare and it is extremely hard to enumerate all of them, it is informative to examine some of them to understand the capabilities of Zipper more comprehensively. Next, let us take two examples to illustrate some of the rare cases where Zipper loses precision.

The no-out flow case. This case is observed in real code in our experiments, and we simplify the code as in Figure 9. The InputStreamReader object (created in line 9) and the FileReader object (created in line 12) flow into the In method BufferedReader (a constructor) through parameter in (line 3). The objects are stored (line 4) and further loaded and become the receiver objects of the virtual call in.close() (line 6). The flow does not flow out through an Out method and thus the two methods in class BufferedReader are analyzed context-insensitively. As a result, the virtual call in line 6 will not be disambiguated into a monomorphic call, resulting in precision loss in the devirtualization analysis client. Note that there would be no observable (in our metrics) precision loss compared to a conventional object-sensitive analysis if the call in line 14 existed (i.e., if it were not commented out). The call site on line 6 is truly polymorphic, and can be exercised for multiple receiver objects, as the addition of line 14 demonstrates.

```
void m(A input,B output) {
  output.field = input;
}
m(a, b);          //rare
b.setField(a);    //common
```

Fig. 10. Example of the parameter-out flow case.

The parameter-out flow case. A second instance where Zipper loses precision, this time made up but interesting theoretically, is shown in Figure 10. An In method m of some class accepts two parameters input and output, and unlike any of our three precision loss patterns, there is no flow out of an Out method. Instead, the flowing-in object through input flows out through another parameter output via a store operation, output.field = input. Thus, Zipper reports m as non-precision-critical. However, if m is analyzed context-insensitively, the flowing-in objects may be merged in the wrapper object (say w, which is pointed to by output) and imprecision would be introduced when the objects are then loaded from w outside

[1]To further validate the generality of Zipper's precision-guided capability, we also compare Zipper with a 2-type-sensitive pointer analysis (2type), another key context sensitivity for Java which is less precise but faster than 2obj. On average, 99.1% of the precision of 2type is preserved by Zipper; the detailed results are shown in Table 6 in Appendix A.

method `m`. This case is rare, since it is unusual in Java programs to modify some field of an object by calling methods such as `m`. In Java, such modification is usually done with a call as in the last line of the example.

4.1.2 How fast is Zipper-guided pointer analysis compared with a conventional analysis? The analysis times for Zipper-guided pointer analysis and `2obj` are shown in the third column in Table 1. On average, Zipper-guided pointer analysis achieves 3.4X speedup compared with `2obj`. The best case is program `xalan` where `2obj` spends about 17 minutes while Zipper-guided analysis finishes running in well under 2 minutes (9.2X speedup). The worst case is program `bloat` where `2obj` spends 52 minutes while Zipper-guided analysis is 7 minutes faster (1.2X speedup).

Recall that the goal of Zipper is not simply to speed up context-sensitive pointer analysis, but to do so while retaining its precision. All methods considered precision-critical are analyzed context-sensitively with the Zipper approach, even though context-insensitive analysis might be faster. This explains the `bloat` case: despite not seeing much efficiency improvement, high precision (98.8%) has been successfully maintained.

Zipper-2obj for bloat.* The strict precision-guided design of Zipper can be relaxed for better efficiency if some heuristics are considered. That is, among the precision-critical methods identified by Zipper, some of them can be further excluded by keeping only the *highly*-precision-critical methods which may cause a significant precision loss if not analyzed context-sensitively. As a proof-of-concept, to identify these highly-precision-critical methods, we simply modify Zipper by adding one more heuristic and apply the modified Zipper (named zipper-2obj* in Table 1) to analyze `bloat` as described below.

The added heuristic is that we do not consider basic flow tracking from an In method unless the flowing-in objects have a large number of different types (for this proof-of-concept experiment, we set the number to 50). As a result, the modified Zipper (zipper-2obj*) reports only 14% of the methods as highly-precision-critical (in comparison, the original Zipper reports 40% of the methods as precision-critical), and the achieved efficiency and precision is shown in the last row of Table 1. The speedup now becomes 60.2X, which is much faster than the original 1.2X; however, as explained above, precision is accordingly hurt: 95.5% of the precision is preserved, which is less than the 98.8% achieved by the original Zipper (zipper-2obj).

This extra experiment demonstrates that heuristic approaches can be developed on top of Zipper via its construction of precision flow graphs. How to make other precision and efficiency trade-offs by leveraging Zipper is not the focus of this paper; however, it may be interesting to explore further in future work.

Note that, as Zipper only reports on average 38% of the methods in a program as precision-critical (see Section 4.3.1), most methods are analyzed context-insensitively, which results in memory savings compared to a conventional context-sensitive pointer analysis. Thus, Zipper is expected to be even more beneficial for memory-constrained analysis environments.

4.1.3 What is the overhead of running Zipper? As shown earlier, in Figure 6, the overhead of Zipper consists of: (1) running a context-insensitive pointer analysis (ci) as Zipper's pre-analysis and (2) running Zipper itself which identifies the precision-critical methods. The analysis time of ci is given in Table 1. On average, ci costs 52 seconds for each program.

Table 2 (last row) shows the performance of Zipper itself: the average analysis time of Zipper is just 32 seconds per input program. Table 2 also lists some related metrics about program size (the number of classes) and elements of Zipper's reasoning, i.e., the number of nodes and edges of the object flow graph (OFG) per program, and the average number of nodes and edges of the

Table 2. Size metrics of all the programs and the corresponding overhead of running Zipper.

Size metrics	batik	checkstyle	sunflow	findbugs	jpc	eclipse	chart	fop	xalan	bloat	avg.
#classes	2 701	1 301	2 496	1 752	2 039	1 122	1 578	2 580	1 268	1 107	1 794
#nodes in OFG	189 993	92 167	188 395	127 939	166 639	84 352	115 515	162 086	94 969	87 223	130 928
#edges in OFG	486 629	203 445	407 526	276 319	386 618	172 161	230 409	370 003	207 300	201 057	294 147
#avg. nodes in PFG	3 576	3 620	3 329	1 711	2 329	1 823	2 513	2 285	3 061	1 942	2 619
#avg. edges in PFG	10 253	9 090	8 677	4 273	6 727	4 480	6 207	6 403	7 478	4 724	6 831
Zipper time (seconds)	102	18	53	13	39	8	16	54	14	8	32

precision flow graph (PFG) per class. The overhead of running Zipper is very small considering the considerable speedup it achieves for costly context-sensitive pointer analysis as shown in Table 1.

4.2 RQ2: Zipper-guided pointer analysis vs. introspective pointer analyses

We next compare Zipper-guided pointer analysis with the most closely related state-of-the-art work: the two introspective analyses, IntroA and IntroB [Smaragdakis et al. 2014], in terms of precision and efficiency.

Detailed comparison results are shown in the last three entries for each program in Table 1. On average, IntroA preserves 74.0% and IntroB keeps 86.5% of the `2obj` precision while Zipper maintains 98.8% of it. Moreover, Zipper achieves better precision than both IntroA and IntroB for *all* four client analyses in *all* the evaluated programs, with the exception of one instance (out of 80): #reach-mtd for `xalan` with IntroB (which is almost 7 times slower than Zipper).

As both Zipper and introspective analysis involve a pre-analysis to select the methods that will be analyzed context-sensitively in the main analysis, the efficiency comparison has two parts: the costs of their pre-analyses and the guided main analyses.

Regarding the pre-analysis, its cost consists of the time of running context-insensitive pointer analysis (for providing basic analysis information) and the time of running Zipper and introspective analysis themselves (for selecting the precision-critical methods). For the former, their costs are the same as they rely on the same context-insensitive pointer analysis provided by Doop. For the latter, for each program, on average, IntroA and IntroB spend 19 and 24 seconds, respectively, while Zipper spends 32 seconds (as shown in Section 4.1.3).

Regarding the main analysis, their results are shown in Table 1 (the third column). In summary, IntroA runs faster than Zipper in 9 out of 10 programs; this comes with no surprise given that the precision of IntroA is only slightly better than context-insensitive analysis while Zipper preserves almost all the precision of a conventional one, i.e., `2obj` in our setting. Zipper runs faster than IntroB in 7 out of 10 programs (except `sunflow`, `findbugs`, and `bloat`) while achieving better precision than IntroB in all cases except #reach-mtd for `xalan`, as described above.

4.3 RQ3: Effect of each precision loss pattern

Zipper identifies precision-critical methods and guides context-sensitive pointer analysis based on the three precision loss patterns introduced in Section 2. In this section, we further evaluate Zipper by measuring the impact of each pattern. We consider four combinations of the three patterns: (1) direct flow alone (`Direct`), (2) direct flow and wrapped flow (`Direct+Wrapped`), (3) direct flow and unwrapped flow (`Direct+Unwrapped`) and (4) all three flows, i.e., Zipper (`Direct+Wrapped+Unwrapped`).

Note that, as direct flow is the basic flow on which wrapped and unwrapped flows depend (Zipper requires direct flow to track the flows of the wrapper and unwrapped objects), the above four combinations cover all reasonable combined cases of the three precision loss patterns.

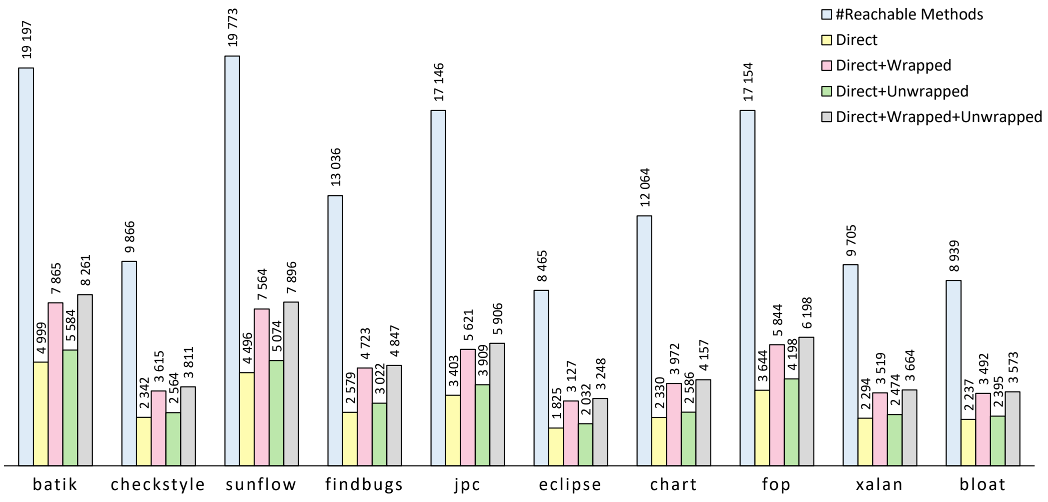

Fig. 11. Precision-critical methods under different combinations of the basic precision loss patterns.

We first evaluate the number of precision-critical methods reported by Zipper under different flow combinations in Section 4.3.1, and then present the precision and efficiency of Zipper-guided pointer analyses with respect to the different flow combinations in Section 4.3.2.

4.3.1 How many methods does Zipper consider precision-critical, and how does each precision loss pattern contribute? Figure 11 gives the numbers of precision-critical methods reported by Zipper under the different combinations of direct, wrapped, and unwrapped flow. `#Reachable Methods` denotes the numbers of methods that are reachable by Zipper's pre-analysis, i.e., a context-insensitive pointer analysis. Let us first focus on `Direct+Wrapped+Unwrapped`, which denotes the combination of all the three patterns and also represents the final results of Zipper. On average, Zipper reports that only 38% of the methods need contexts per program under `Direct+Wrapped+Unwrapped`. As shown in Section 4.1.1, applying context sensitivity to only this 38% of the methods is able to preserve 98.8% of the precision of conventional 2-object-sensitive pointer analysis.

In Figure 11, we can see that Zipper reports that 22.3% of the methods need contexts under `Direct`, 36.4% under `Direct+Wrapped`, and 24.9% under `Direct+Unwrapped`, which shows that wrapped flow introduces significantly more precision-critical methods than unwrapped flow. `Direct+Unwrapped` introduces 2.6% more methods than `Direct`, while `Direct+Wrapped+Unwrapped` introduces 1.6% more methods than `Direct+Wrapped`. This means that some methods are involved in multiple precision loss patterns, e.g., both wrapped flow and unwrapped flow, simultaneously.

4.3.2 How does each precision loss pattern affect the precision and efficiency of Zipper-guided pointer analysis? We evaluate the impact of each precision loss pattern by using Zipper with different combinations of patterns to guide `2obj` analysis.

Precision. To evaluate the precision of `2obj` under Zipper's different elements, we focus on the #poly-call metric as it is one of the most representative metrics and also widely considered in Java pointer analysis research [Jeong et al. 2017; Kastrinis and Smaragdakis 2013; Lhoták and Hendren 2006; Smaragdakis et al. 2011, 2014; Sridharan et al. 2005; Tan et al. 2017]. It denotes the number of virtual calls that cannot be disambiguated into monomorphic calls. Generally, a pointer analysis with better precision can disambiguate more virtual calls and reports smaller #poly-call.

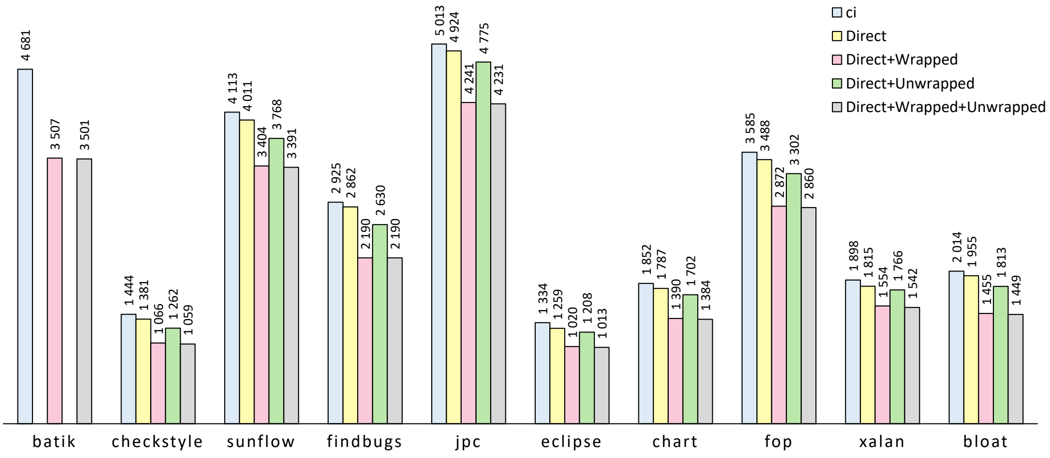

Fig. 12. #poly-call for different combinations of the basic precision loss patterns.

Figure 12 shows #poly-call as reported by the Zipper-guided pointer analyses under different combinations of direct, wrapped, and unwrapped flow. We use the #poly-call reported by the context-insensitive pointer analysis (denoted by ci) as the baseline. Overall, Zipper with more flow patterns enabled achieves better precision. (`batik` lacks data for `Direct` and `Direct+Unwrapped` since the pointer analysis cannot terminate within the time budget under these two combinations; the reason will be discussed later.)

The direct flow pattern covers the usage of simple object flow (e.g., *getter*/*setter* methods), which is common in Java programs. However, Figure 12 shows, perhaps surprisingly, that Zipper under `Direct` is only slightly more precise than context-insensitive pointer analysis. These results demonstrate that only applying context sensitivity to the methods involved in direct flow is far from sufficient for achieving good precision.

When wrapped flow comes into play, the precision is improved significantly. For example, compared to `Direct`, Zipper under `Direct+Wrapped` further eliminates 683 false polymorphic calls for `jpc`, and this improvement is much greater than that of `Direct` compared to ci (89 calls). The data for other programs exhibit similar trends, which means that wrapped flow is the key to preserving the precision of conventional object-sensitivity.

Unwrapped flow is also useful for improving precision. For example, for `sunflow`, Zipper under `Direct+Unwrapped` eliminates 243 false polymorphic calls based on `Direct`. However, the improvements of unwrapped flow become less significant after combining with wrapped flow. For example, for `sunflow`, Zipper under `Direct+Wrapped+Unwrapped` only eliminates 13 false polymorphic calls based on `Direct+Wrapped`. One reason is that some precision-critical methods introduced by unwrapped flow can also be introduced by wrapped flow, as discussed in Section 4.3.1.

Efficiency. Table 3 gives the elapsed time of Zipper-guided pointer analysis under different combinations of the three precision loss patterns. Generally, when more patterns are enabled, Zipper reports more methods as precision-critical, and the corresponding guided pointer analysis run faster. For all programs, Zipper under `Direct+Wrapped` runs faster than `Direct` alone, and for 6 out of 10 programs, Zipper under `Direct+Wrapped+Unwrapped` runs faster than `Direct+Wrapped`.

These results clearly demonstrate that losing precision may also introduce performance decline. This is especially typical for context-sensitive pointer analysis, as the spurious data flow (caused by imprecision) will be replicated and propagated under different contexts, which can make the

Table 3. The corresponding performance (seconds) of the analyses in Figure 12.

	batik	checkstyle	sunflow	findbugs	jpc	eclipse	chart	fop	xalan	bloat
Direct	–	477	1 492	2 298	1 851	127	433	2 751	262	3 119
Direct+Wrapped	873	287	500	841	221	71	82	427	122	2 764
Direct+Unwrapped	–	458	2 720	2 952	3 454	117	632	5 147	251	3 108
Direct+Wrapped+Unwrapped	927	355	520	830	211	66	77	457	107	2 704

pointer analysis very inefficient. For example, Zipper under `Direct` and `Direct+Unwrapped` is less precise than `Direct+Wrapped+Unwrapped`. While the first two analyses cannot even finish within the time budget (1.5 hours) for `batik`, the last one requires just 927 seconds.

4.4 Robustness of Zipper

To further test the robustness of Zipper, we show the results of two supplementary experiments.

Different benchmark programs. First, we show Zipper's effectiveness over the available benchmarks from the DaCapo 2009 set (instead of the original DaCapo 2006 benchmarks). The DaCapo 2009 benchmarks are less commonly used in static analysis research, since they present some engineering complications. The benchmarks do not provide stub classes for each benchmark's individual execution and are instead driven by code that employs Java reflection. To overcome such complications, we use the supportive log files (the Tamiflex [Bodden et al. 2011] reflection logs) and the packed jars (to enable whole-program static analysis) from the current Doop project repository. We consider only 4 benchmarks for this experiment: `xalan`, `avrora`, `batik` and `sunflow`. Of the rest, `fop`, `tomcat`, `tradesoap` are not considered as their log files or packed jars are not available in the repository; `lusearch`, `luindex` and `pmd` are not considered, just as in DaCapo 2006, as they are trivially scalable; `jython`, `eclipse`, `h2` and `tradebeans` are not considered as the baseline analysis (`2obj`) cannot finish running even in three hours (as opposed to the default 1.5 hours of our previous experiment).

Table 4 shows the results for the four available DaCapo 2009 benchmarks. On average, Zipper achieves 99.5% of the precision of `2obj` with a speedup of 3.0X. In addition, IntroB is both less

Table 4. Performance and precision metrics of different analyses on the available DaCapo 2009 benchmarks.

Program	Pointer analysis	Time (s)	#fail-cast	#poly-call	#reach-mtd	#call-edge
xalan09	ci	58	1 975	3 722	13 386	75 626
	2obj	1 257	1 054	3 089	12 957	66 601
	zipper-2obj	283	1 074	3 092	12 962	66 652
	introA-2obj	159	1 779	3 507	13 288	74 275
	introB-2obj	444	1 395	3 228	13 086	68 891
avrora09	ci	62	1 829	2 141	15 244	71 656
	2obj	267	1 026	1 557	14 752	61 742
	zipper-2obj	156	1 042	1 560	14 755	61 766
	introA-2obj	146	1 659	1 858	15 128	69 994
	introB-2obj	304	1 381	1 651	14 920	65 762
batik09	ci	118	3 651	5 951	22 075	128 940
	2obj	8 295	2 234	5 172	21 464	112 800
	zipper-2obj	2 254	2 268	5 178	21 467	112 834
	introA-2obj	370	3 383	5 531	21 866	123 538
	introB-2obj	831	2 903	5 309	21 684	118 533
sunflow09	ci	48	2 099	2 504	14 120	70 733
	2obj	208	1 179	1 948	13 576	60 402
	zipper-2obj	95	1 192	1 955	13 592	60 460
	introA-2obj	102	1 849	2 262	14 014	69 269
	introB-2obj	168	1 558	2 046	13 746	64 045

Table 5. Precision metrics of different analyses on the trivially-scalable DaCapo benchmarks.

Program	Pointer analysis	#fail-cast	#poly-call	#reach-mtd	#call-edge	Program	Pointer analysis	#fail-cast	#poly-call	#reach-mtd	#call-edge
antlr	ci	992	1 776	7 794	53 468	lusearch09	ci	925	1 373	8 035	40 968
	2obj	428	1 520	7 357	49 348		2obj	351	1 056	7 701	36 227
	zipper-2obj	452	1 530	7 361	49 400		zipper-2obj	366	1 061	7 705	36 254
	introA-2obj	990	1 694	7 783	53 071		introA-2obj	756	1 184	7 934	39 673
	introB-2obj	640	1 560	7 448	50 257		introB-2obj	536	1 101	7 764	37 164
lusearch	ci	844	1 133	7 352	36 343	luindex09	ci	919	1 239	8 144	41 622
	2obj	299	850	6 904	31 811		2obj	378	851	7 824	36 886
	zipper-2obj	322	864	6 907	31 869		zipper-2obj	399	855	7 827	36 911
	introA-2obj	681	981	7 277	35 531		introA-2obj	785	1 000	8 024	40 106
	introB-2obj	462	891	6 970	32 656		introB-2obj	586	919	7 891	37 881
luindex	ci	734	940	6 670	33 130	pmd09	ci	1 514	1 571	9 770	49 388
	2obj	297	675	6 256	29 021		2obj	877	1 114	9 431	43 735
	zipper-2obj	327	686	6 259	29 076		zipper-2obj	898	1 124	9 434	43 770
	introA-2obj	617	802	6 600	32 370		introA-2obj	1 383	1 277	9 670	47 597
	introB-2obj	450	714	6 316	29 835		introB-2obj	1 144	1 195	9 527	45 125
pmd	ci	1 263	1 039	8 427	42 415						
	2obj	657	718	7 648	35 563						
	zipper-2obj	676	728	7 654	35 626						
	introA-2obj	1 136	882	8 351	41 674						
	introB-2obj	859	777	7 929	37 379						

precise and less efficient than Zipper in most cases, and IntroA runs faster but is significantly less precise than Zipper in all cases. These results are consistent with the ones for the DaCapo 2006 benchmarks and other real-world Java applications as reported in Section 4.1.

Precision for "trivially-scalable" programs. We also evaluate Zipper's precision for those "trivially-scalable" DaCapo 2006 and DaCapo 2009 benchmarks that were excluded from our earlier presentation. Although Zipper would likely not be used for such programs (since a highly-precise pointer analysis can already analyze them very fast), it is interesting to ask if it still maintains most of the precision of a highly-precise context-sensitive analysis (i.e., `2obj`) for these programs.

Table 5 shows the precision results of Zipper for the seven trivially-scalable DaCapo (2006 and 2009) benchmarks (note that the DaCapo 2009 benchmark suite does not contain `antlr`). The results demonstrate that Zipper is able to preserve most of the precision (98.3% on average) of `2obj` even for those trivially-scalable programs that are outside the target domain of Zipper.

5 RELATED WORK

In this section, we mainly discuss related work that leverages pre-analysis to achieve good precision and efficiency balances for whole-program context-sensitive pointer analysis.

Introspective analysis [Smaragdakis et al. 2014] applies context sensitivity to a subset of the program's methods selected based on two heuristics, resulting in two introspective analyses, IntroA and IntroB, which have been compared with Zipper in Section 4.2. Like Zipper, introspective analysis first performs a cheap pre-analysis, i.e., a context-insensitive pointer analysis, to extract required information to guide the main pointer analysis. Unlike Zipper, it relies on a set of six manually-selected metrics (e.g., the cumulative size of points-to set over all local variables of each method) to define the two heuristics for determining which methods are potentially precision-critical. As these heuristics lack a theoretical explanation of when omitting context sensitivity for a method would introduce imprecision, the precision-critical methods cannot be identified accurately by introspective analysis. As a result, as shown in Section 4.2, IntroB is less precise and less efficient than Zipper in most cases, and IntroA runs faster but is significantly less precise than Zipper in all cases.

Hassanshahi et al. [2017] also leverage manually-selected metrics to define some heuristics to guide object-sensitive pointer analysis for large codebases. Their pre-analysis contains several phases that each need different metrics and heuristics. Basically, a program kernel (where a call-site-insensitive or object-sensitive pointer analysis may not be precise enough) is first extracted based on a context-insensitive pointer analysis, and then this kernel is analyzed by a fixed object-sensitive pointer analysis to determine the appropriate context depth for each selected object. Such information is finally used to guide a selective object-sensitive pointer analysis, which has been demonstrated to work well for the OpenJDK library [Hassanshahi et al. 2017]. However, unlike introspective analysis [Smaragdakis et al. 2014] and Zipper, the overhead of their pre-analysis is uncertain, as it is sensitive to the complexity of the extracted kernel, which further depends on various threshold values given by the user before the pre-analysis.

Different from introspective analysis and the approach by Hassanshahi et al. [2017], Zipper does not rely on any inputs (i.e., various threshold values needed by heuristics) provided by users. Instead, Zipper's precision-guided principle enables it to identify the precision-critical methods by exploiting the precision loss patterns only from the programs themselves. As a result, Zipper can exhibit more stable analysis results.

Metrics and heuristics can be selected and defined manually, as in the above approaches [Hassanshahi et al. 2017; Smaragdakis et al. 2014], or can be learned from machine learning techniques, as in the two pieces of work we describe next.

Wei and Ryder [2015] introduce an adaptive context-sensitive analysis for JavaScript. Some user-specific method features are first extracted from an inexpensive pre-analysis, and a machine learning algorithm is then applied to obtain the relationship between these method features and the potential context-sensitivity candidates. The relationship is expressed as a decision tree, which is further manually adjusted (based on domain knowledge) to produce certain heuristics. Guided by these heuristics, different methods are finally analyzed with different context sensitivity.

Jeong et al. [2017] present a data-driven approach to guiding context-sensitive analysis for Java. Unlike introspective analysis and Zipper, where for each method, context sensitivity is either applied or not, the data-driven analysis assigns each method an appropriate context length including zero (i.e., context insensitivity). By appropriately applying context sensitivity with deeper context for only a subset of the methods, more efficient context-sensitive analysis can be achieved with good precision. To assign an appropriate context length for each method, 25 metrics (atomic features) are selected, and, based on these metrics, a machine learning approach is used to learn heuristics. However, unlike Zipper's lightweight pre-analysis, the learning phase is heavy and costs 54 hours in Jeong et al.'s experimental setting. Still, the learned heuristics can help the main analysis scale for even some trouble programs (e.g., `jython`) with good precision [Jeong et al. 2017]. One reason that may contribute to the beneficial effect of the learned heuristics is that the training programs and the testing programs partly share the same Java library code.

Generally, machine learning approaches are sensitive to the training process on the selected input programs, and the learned results are usually difficult to explain, e.g., why the learning algorithm considers method A rather than B as precision-critical. Differently, Zipper is a principled approach derived from the insight of exploiting the precision loss patterns inherent in a program; thus its guiding is interpretable and its guided results are tractable, resulting in more uniform and stable effectiveness achieved.

The Bean approach by Tan et al. [2016] is also based on a pre-analysis. Conventional context-sensitivity uses consecutive context elements for each context, whereas Bean identifies and skips context elements that are useless for improving the precision. As a result, more space is saved and, thus, more precision-useful context elements can be added to distinguish more contexts, making the pointer analysis more precise with a small efficiency overhead. Instead of improving precision by sacrificing some efficiency, Zipper makes a context-sensitive pointer analysis run faster while preserving essentially all of its precision.

Scaler [Li et al. 2018] achieves scalable context-sensitive points-to analysis by considering the relationship between scalability and memory size. It leverages the object allocation graph (OAG) proposed by Tan et al. [2016], to efficiently estimate the amount of context-sensitive points-to information that would be needed for each method. Then, given a threshold related to the available memory size, Scaler selects an appropriate context-sensitivity variant for each method so that the total amount of points-to information is bounded. As a result, Scaler utilizes the available space to provide scalability while maximizing precision. Unlike Zipper which prioritizes precision, Scaler is a scalability-first approach. The two techniques can be combined, using Scaler to estimate the context-sensitive points-to information only for the precision-critical methods identified by Zipper.

Based on a cheap pre-analysis, Tan et al. [2017] present Mahjong, a heap abstraction for pointer analysis of Java, which enables allocation-site-based pointer analysis to run significantly faster while achieving almost the same precision for type-dependent clients, such as call graph construction. Differently, Zipper works for general pointer analysis, including alias analysis (i.e., not just type-dependent clients), which cannot be handled effectively by Mahjong.

Bean [Tan et al. 2016] and Scaler [Li et al. 2018] leverage object allocation graphs (OAGs), and Mahjong [Tan et al. 2017] exploits field points-to graphs (FPGs), in a pre-analysis to extract necessary information to guide a later main analysis. Similarly, in Zipper, we introduce precision flow graphs (PFGs) to express the three kinds of value flow patterns (Section 2) and identify the precision-critical methods by solving a graph reachability problem on the PFG (Section 3.3). OAGs and FPGs cannot express value flow information and are therefore conceptually different from PFGs. However, other graphs, conceptually similar to PFGs, are used in pointer analysis, as briefly discussed next.

Li et al. [2011] leverage value flow graphs (VFGs) to accelerate pointer analysis for C/C++ programs. VFGs are also designed to express value flow information but they have two key differences from PFGs. First, they represent pointer information differently, e.g., dereferencing a pointer in C/C++ does not involve a field reference. Second, VFGs cannot express wrapped/unwrapped flows.

Pointer assignment graphs (PAGs) are used as the representation of the analyzed program in Java pointer analysis [Lhoták and Hendren 2003]. A field reference node in a PAG is a field dereference on a variable while an object field node in a PFG is a field dereference on the object pointed to by a variable. Thus, unlike PFGs, the value flow through load/store operations is not connected in PAGs, e.g., given statements `p.f = a` and `b = q.f`, there is no path from `a` to `b` in the PAG even if variables `p` and `q` point to the same object. Therefore, unlike PFGs, PAGs cannot express the flow of objects in a program directly.

Recent research has produced efficient *demand-driven* pointer analyses (e.g., [Späth et al. 2016; Wang et al. 2017]). A demand-driven analysis typically only computes points-to information for program points that may affect a particular site of interest for specific clients. Differently, the Zipper analysis and other whole-program pointer analyses [Hassanshahi et al. 2017; Jeong et al. 2017; Smaragdakis et al. 2014; Tan et al. 2016, 2017] compute points-to information for all sites, thereby providing information for all possible clients.

6 CONCLUSION

Context sensitivity is an important technique for ensuring high precision in pointer analysis for Java. Previous work has shown that it is beneficial to apply context sensitivity selectively, instead of uniformly for all methods, as conventionally done. In this paper, we have presented Zipper: a principled approach to identifying precision-critical methods, with a focus on keeping as much precision as possible compared to a conventional analysis.

The conceptual contribution of this work consists of the three basic patterns of value flows (direct, wrapped, and unwrapped flows) that explain where and how most imprecision is introduced in a context-insensitive pointer analysis, together with the concept of precision flow graphs that concisely model the relevant value flow. The practical contribution consists of the implementation and experiments, which demonstrate the effectiveness of the technique on real-world Java programs. The experimental results show that the three precision loss patterns successfully capture the vast majority of the methods that benefit from context sensitivity, and as a result, we obtain a significant analysis speedup while retaining essentially all of the precision of conventional context-sensitive pointer analysis. Zipper is conceptually simple and easy to integrate with existing pointer analysis tools.

For future work, it is interesting to further explore the opportunities for relaxing the precision-guided principle (as suggested in Section 4.1.2), and to use the Zipper approach at a more fine-grained level, on variables and object fields instead of methods (as mentioned in Section 3.3).

A APPENDIX

Table 6. Performance and precision metrics for context-insensitive (ci), conventional type-sensitive (2type), Zipper-guided (zipper-2type), and introspective type-sensitive (introX-2type) pointer analyses.

Program	Pointer analysis	Time (s)	#fail-cast	#poly-call	#reach-mtd	#call-edge
batik	ci	82	2 961	4 681	19 197	101 616
	2type	378	1 938	3 623	16 892	77 337
	zipper-2type	239	1 941	3 617	16 894	77 351
	introA-2type	187	2 751	4 316	19 027	97 330
	introB-2type	509	2 398	4 107	18 733	90 824
checkstyle	ci	50	1 114	1 444	9 866	57 490
	2type	125	695	1 122	9 534	49 274
	zipper-2type	82	711	1 140	9 544	49 436
	introA-2type	106	982	1 267	9 785	55 876
	introB-2type	156	852	1 205	9 613	51 718
sunflow	ci	61	3 003	4 113	19 773	106 410
	2type	197	2 247	3 506	19 315	90 967
	zipper-2type	136	2 262	3 510	19 316	91 022
	introA-2type	126	2 840	3 855	19 685	104 135
	introB-2type	169	2 561	3 635	19 476	96 376
findbugs	ci	52	2 508	2 925	13 036	77 370
	2type	265	1 683	2 345	12 674	66 443
	zipper-2type	179	1 703	2 349	12 678	66 488
	introA-2type	120	2 296	2 581	12 987	74 820
	introB-2type	225	2 068	2 534	12 901	72 006
jpc	ci	58	2 370	5 013	17 146	96 669
	2type	128	1 599	4 328	15 908	81 527
	zipper-2type	98	1 614	4 336	15 911	81 559
	introA-2type	117	2 224	4 776	17 063	95 417
	introB-2type	156	1 868	4 413	16 046	86 709
eclipse	ci	23	1 139	1 334	8 465	45 474
	2type	57	665	1 031	7 933	38 337
	zipper-2type	50	714	1 063	7 967	38 677
	introA-2type	55	1 004	1 161	8 336	43 911
	introB-2type	63	850	1 100	8 026	40 289
chart	ci	46	1 810	1 852	12 064	63 453
	2type	84	1 155	1 446	11 439	52 965
	zipper-2type	79	1 175	1 451	11 444	53 011
	introA-2type	108	1 664	1 658	11 976	61 731
	introB-2type	121	1 384	1 541	11 579	56 380
fop	ci	74	2 458	3 585	17 154	84 330
	2type	251	1 753	2 930	16 477	71 847
	zipper-2type	189	1 777	2 943	16 482	71 922
	introA-2type	199	2 288	3 298	17 024	82 301
	introB-2type	278	2 005	3 045	16 618	76 638
xalan	ci	39	1 182	1 898	9 705	51 302
	2type	99	729	1 565	9 151	45 444
	zipper-2type	79	758	1 578	9 160	45 566
	introA-2type	107	1 136	1 793	9 655	50 775
	introB-2type	130	889	1 640	9 232	46 927
bloat	ci	31	1 924	2 014	8 939	61 150
	2type	74	1 486	1 626	8 523	54 279
	zipper-2type	73	1 508	1 642	8 536	54 424
	introA-2type	61	1 833	1 812	8 885	60 305
	introB-2type	73	1 714	1 684	8 647	56 041

ACKNOWLEDGMENTS

This work was supported by the European Research Council (ERC) under the FP7 and Horizon 2020 research and innovation programs (grant agreements 307334, 790340, and 647544).

REFERENCES

Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language.* Ph.D. Dissertation. University of Copenhagen.

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 259–269. https://doi.org/10.1145/2594291.2594299

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, Peri L. Tarr and William R. Cook (Eds.). ACM, 169–190. https://doi.org/10.1145/1167473.1167488

Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, May 21-28, 2011.* 241–250. https://doi.org/10.1145/1985793.1985827

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, Shail Arora and Gary T. Leavens (Eds.). ACM, 243–262. https://doi.org/10.1145/1640089.1640108

Satish Chandra, Stephen J. Fink, and Manu Sridharan. 2009. Snugglebug: a powerful approach to weakest preconditions. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 363–374. https://doi.org/10.1145/1542476.1542517

David R. Chase, Mark N. Wegman, and F. Kenneth Zadeck. 1990. Analysis of Pointers and Structures. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990*, Bernard N. Fischer (Ed.). ACM, 296–310. https://doi.org/10.1145/93542.93585

Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2008. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.* 17, 2 (2008), 9:1–9:34. https://doi.org/10.1145/1348250.1348255

Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. 2015. Information Flow Analysis of Android Applications in DroidSafe. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015.* The Internet Society. https://www.ndss-symposium.org/ndss2015/information-flow-analysis-android-applications-droidsafe

Neville Grech and Yannis Smaragdakis. 2017. P/Taint: unified points-to and taint analysis. *PACMPL* 1, OOPSLA (2017), 102:1–102:28. https://doi.org/10.1145/3133926

Behnaz Hassanshahi, Raghavendra Kagalavadi Ramesh, Padmanabhan Krishnan, Bernhard Scholz, and Yi Lu. 2017. An efficient tunable selective points-to analysis for large codebases. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2017, Barcelona, Spain, June 18, 2017*, Karim Ali and Cristina Cifuentes (Eds.). ACM, 13–18. https://doi.org/10.1145/3088515.3088519

Michael Hind. 2001. Pointer analysis: haven't we solved this problem yet?. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'01, Snowbird, Utah, USA, June 18-19, 2001*, John Field and Gregor Snelting (Eds.). ACM, 54–61. https://doi.org/10.1145/379605.379665

Sehun Jeong, Minseok Jeon, Sung Deok Cha, and Hakjoo Oh. 2017. Data-driven context-sensitivity for points-to analysis. *PACMPL* 1, OOPSLA (2017), 100:1–100:28. https://doi.org/10.1145/3133924

Vini Kanvar and Uday P. Khedker. 2016. Heap Abstractions for Static Analysis. *ACM Comput. Surv.* 49, 2, Article 29 (June 2016), 47 pages. https://doi.org/10.1145/2931098

George Kastrinis and Yannis Smaragdakis. 2013. Hybrid context-sensitivity for points-to analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 423–434. https://doi.org/10.1145/2462156.2462191

Ondrej Lhoták and Laurie J. Hendren. 2003. Scaling Java Points-to Analysis Using SPARK. In *Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings (Lecture Notes in Computer Science)*, Görel Hedin (Ed.), Vol. 2622.

Springer, 153–169. https://doi.org/10.1007/3-540-36579-6_12

Ondrej Lhoták and Laurie J. Hendren. 2006. Context-Sensitive Points-to Analysis: Is It Worth It?. In *Compiler Construction, 15th International Conference, CC 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 30-31, 2006, Proceedings (Lecture Notes in Computer Science)*, Alan Mycroft and Andreas Zeller (Eds.), Vol. 3923. Springer, 47–64. https://doi.org/10.1007/11688839_5

Lian Li, Cristina Cifuentes, and Nathan Keynes. 2011. Boosting the Performance of Flow-sensitive Points-to Analysis Using Value Flow. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, New York, NY, USA, 343–353. https://doi.org/10.1145/2025113.2025160

Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Scalability-First Pointer Analysis with Self-Tuning Context-Sensitivity. In *Proc. 12th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

Yue Li, Tian Tan, Yifei Zhang, and Jingling Xue. 2016. Program Tailoring: Slicing by Sequential Criteria. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy (LIPIcs)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.), Vol. 56. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 15:1–15:27. https://doi.org/10.4230/LIPIcs.ECOOP.2016.15

Benjamin Livshits and Monica S. Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005*, Patrick D. McDaniel (Ed.). USENIX Association.

Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2002. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002*, Phyllis G. Frankl (Ed.). ACM, 1–11. https://doi.org/10.1145/566172.566174

Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.* 14, 1 (2005), 1–41. https://doi.org/10.1145/1044834.1044835

Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, Michael I. Schwartzbach and Thomas Ball (Eds.). ACM, 308–319. https://doi.org/10.1145/1133981.1134018

Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. 2009. Effective static deadlock detection. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, 386–396. https://doi.org/10.1109/ICSE.2009.5070538

Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. 2014. Selective Context-sensitivity Guided by Impact Pre-analysis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 475–484. https://doi.org/10.1145/2594291.2594318

Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R. Gross. 2012. Statically checking API protocol conformance with mined multi-object specifications. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE Computer Society, 925–935. https://doi.org/10.1109/ICSE.2012.6227127

Barbara G. Ryder. 2003. Dimensions of Precision in Reference Analysis of Object-Oriented Programming Languages. In *Compiler Construction, 12th International Conference, CC (Lecture Notes in Computer Science)*, Vol. 2622. Springer, 126–137.

Bernhard Scholz, Herbert Jordan, Pavle Subotic, and Till Westmann. 2016. On fast large-scale program analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, Ayal Zaks and Manuel V. Hermenegildo (Eds.). ACM, 196–206. https://doi.org/10.1145/2892208.2892226

Micha Sharir and Amir Pnueli. 1981. *Two Approaches to Interprocedural Data Flow Analysis.* Chapter 7, 189–233.

Olin Shivers. 1991. *Control-flow analysis of higher-order languages.* Ph.D. Dissertation. Carnegie Mellon University.

Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Foundations and Trends in Programming Languages* 2, 1 (2015), 1–69. https://doi.org/10.1561/2500000014

Yannis Smaragdakis, George Balatsouras, and George Kastrinis. 2013. Set-based Pre-processing for Points-to Analysis. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications (OOPSLA '13)*. ACM, New York, NY, USA, 253–270. https://doi.org/10.1145/2509136.2509524

Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 17–30. https://doi.org/10.1145/1926385.1926390

Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective analysis: context-sensitivity, across the board. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 485–495. https://doi.org/10.1145/2594291.2594320

Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *30th European Conference on Object-Oriented Programming, ECOOP 2016,* *July 18-22, 2016, Rome, Italy*. 22:1–22:26. https://doi.org/10.4230/LIPIcs.ECOOP.2016.22

Manu Sridharan and Rastislav Bodík. 2006. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, Michael I. Schwartzbach and Thomas Ball (Eds.). ACM, 387–400. https://doi.org/10.1145/1133981.1134027

Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. 2013. Alias Analysis for Object-Oriented Programs. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). Lecture Notes in Computer Science, Vol. 7850. Springer, 196–232. https://doi.org/10.1007/978-3-642-36946-9_8

Manu Sridharan, Stephen J. Fink, and Rastislav Bodík. 2007. Thin slicing. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, Jeanne Ferrante and Kathryn S. McKinley (Eds.). ACM, 112–122. https://doi.org/10.1145/1250734.1250748

Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven points-to analysis for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, Ralph E. Johnson and Richard P. Gabriel (Eds.). ACM, 59–76. https://doi.org/10.1145/1094811.1094817

Tian Tan, Yue Li, and Jingling Xue. 2016. Making k-Object-Sensitive Pointer Analysis More Precise with Still k-Limiting. In *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings (Lecture Notes in Computer Science)*, Xavier Rival (Ed.), Vol. 9837. Springer, 489–510. https://doi.org/10.1007/978-3-662-53413-7_24

Tian Tan, Yue Li, and Jingling Xue. 2017. Efficient and precise points-to analysis: modeling the heap by merging equivalent automata. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 278–291. https://doi.org/10.1145/3062341.3062360

Rei Thiessen and Ondřej Lhoták. 2017. Context Transformations for Pointer Analysis. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 263–277. https://doi.org/10.1145/3062341.3062359

Paolo Tonella and Alessandra Potrich. 2005. *Reverse Engineering of Object Oriented Code*. Springer. https://doi.org/10.1007/b102522

Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, November 8-11, 1999, Mississauga, Ontario, Canada*, Stephen A. MacKay and J. Howard Johnson (Eds.). IBM, 13. https://doi.org/10.1145/781995.782008

WALA. 2018. Watson Libraries for Analysis. http://wala.sf.net.

Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. 2017. Graspan: A Single-machine Disk-based Graph System for Interprocedural Static Analyses of Large-scale Systems Code. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 389–404. https://doi.org/10.1145/3037697.3037744

Shiyi Wei and Barbara G. Ryder. 2015. Adaptive Context-sensitive Analysis for JavaScript. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic (LIPIcs)*, John Tang Boyland (Ed.), Vol. 37. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 712–734. https://doi.org/10.4230/LIPIcs.ECOOP.2015.712

Guoqing Xu and Atanas Rountev. 2008. Merging Equivalent Contexts for Scalable Heap-cloning-based Context-sensitive Points-to Analysis. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA '08)*. ACM, New York, NY, USA, 225–236. https://doi.org/10.1145/1390630.1390658