A Principled Approach to Selective Context Sensitivity for Pointer Analysis

YUE LI, Nanjing University, China and Aarhus University, Denmark
TIAN TAN*, Nanjing University, China and 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 static 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. Using this theoretical foundation, we present an efficient algorithm, Zipper, to recognize these flow patterns in a given program and employ context sensitivity accordingly. We also present a variant, Zipper^e, that additionally takes into account which methods are disproportionally costly to analyze with context sensitivity.

Our experimental results on standard benchmark and real-world Java programs show that Zipper preserves effectively all of the precision (98.8%) of a highly-precise conventional context-sensitive pointer analysis (2-object-sensitive with a context-sensitive heap, `2obj` for short), with a substantial speedup (on average, 3.4× and up to 9.4×), and that Zipper^e preserves 94.7% of the precision of `2obj`, with an order-of-magnitude speedup (on average, 25.5× and up to 88×). In addition, for 10 programs that cannot be analyzed by `2obj` within a 3 hours time limit, on average Zipper^e can guide `2obj` to finish analyzing them in less than 11 minutes with high precision compared to context-insensitive and introspective context-sensitive analyses.

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. 2020. A Principled Approach to Selective Context Sensitivity for Pointer Analysis. *ACM Trans. Program. Lang. Syst.* 1, 1, Article 1 (January 2020), 40 pages. https://doi.org/10.1145/3381915

1 INTRODUCTION

Pointer analysis is a fundamental family of static analyses that compute abstractions of 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

*Corresponding author

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2020 Association for Computing Machinery.
0164-0925/2020/1-ART1 $15.00
https://doi.org/10.1145/3381915

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, 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 *precision-critical methods* has been the focus of past work [Hassanshahi et al. 2017; Jeon et al. 2019; 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 article, we provide a new 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.

Additionally, we provide a variant of Zipper, named Zipper^e, which also identifies *efficiency-critical methods*, i.e., methods that are costly to analyze with context sensitivity. With Zipper^e, only the methods that are precision-critical but not efficiency-critical will be analyzed context-sensitively. By applying context sensitivity only to the methods selected by Zipper or Zipper^e, a pointer analysis runs significantly faster than conventional techniques that apply context sensitivity indiscriminately to all methods, while retaining most of the attainable precision.

In summary, we make the following key contributions.

- We describe three general patterns of value flow that help in explaining how and where most of the imprecision is introduced in a context-insensitive pointer analysis (Section 3).
- 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 4).

Zipper can guide context-sensitive pointer analysis to run faster while keeping most of its precision.

- We propose the Zipper *express* (Zipper^e) approach, which is developed on the basis of Zipper but additionally considers analysis cost when selecting the methods to be analyzed with context sensitivity (Section 5). Generally, Zipper^e can guide context-sensitive pointer analysis to run extremely fast while being reasonably precise.

  In contrast to other techniques that apply context sensitivity selectively, the Zipper and Zipper^e approaches are based on a tangible understanding of imprecision and not on heuristics that require non-transparent machine learning or other tuning of multiple and complex analysis parameters.
- We provide an extensive experimental evaluation to evaluate the effectiveness of Zipper and Zipper^e (Section 6).
  - 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 (`2obj`), for a speedup of 3.4× and up to 9.2×. These results demonstrate that the three general patterns of value flows indeed capture the vast majority of methods that benefit from context sensitivity.
  - On average, Zipper^e reports that only 14% of the methods are precision-critical but not efficiency-critical, which preserves 94.7% of the precision for `2obj` for a speedup of 25.5× and up to 88×. In addition, for 10 programs that `2obj` is unable to analyze within 3 hours, on average, Zipper^e can guide `2obj` to finish analyzing them under 11 minutes with still good precision. Moreover, for some programs, Zipper^e-guided pointer analysis can even run faster while being more precise than context-insensitive analysis. These results establish Zipper^e as a new sweet spot in state-of-the-art pointer analyses, for making highly practical trade-offs between efficiency and precision.

The present article extends and supersedes the paper by Li et al. [2018a] presented at the *ACM Conference on Object-Oriented Programming, Languages, Systems, and Applications 2018 (OOPSLA 2018)*. In comparison, this article contains the following major extensions (and reflects an updated, more complete understanding throughout the rest of the text):

- We add a brief literature review of the history and current trends in context-sensitive pointer analysis for Java (Section 2).
- We present the new pointer analysis variant, Zipper^e (Zipper *express*), which is extremely fast with also good precision (Section 5).
- We investigate whether the precision-loss patterns of Zipper, as its theoretical foundation, are general enough to effectively identify the precision-critical methods and thus preserve the precision for other mainstream context-sensitivity variants, including call-site sensitivity, type sensitivity and hybrid context sensitivity (Section 6.5).
- We conduct new experiments and extensively evaluate Zipper^e in terms of efficiency and precision in practice (Section 6.6).

2 CONTEXT SENSITIVITY: A BRIEF REVIEW

In this section, we describe how context-sensitivity has evolved for Java pointer analysis from purely improving precision to making good precision and efficiency trade-offs. In addition to giving a brief review to discuss this important and intricate research topic, we also provide some necessary background for our work. We focus on whole-program analysis, which is the main application setting of this research topic [Smaragdakis and Balatsouras 2015; Sridharan et al. 2013].

There are two major factors that control the precision of pointer analysis: how to abstract the heap, and how to abstract the call stack. Compared with the few approaches on heap abstraction for Java pointer analysis [Kanvar and Khedker 2016; Lhoták and Hendren 2003; Tan et al. 2017], there is ample literature on stack modeling. Among the schemes for modeling the stack, in the past years, context sensitivity has been considered as the most effective approach to improve the analysis precision for Java programs [Lhoták and Hendren 2006] and has thus attracted the most attention from researchers [Bravenboer and Smaragdakis 2009; Hassanshahi et al. 2017; Jeon et al. 2019, 2018; Jeong et al. 2017; Kastrinis and Smaragdakis 2013; Lhoták and Hendren 2006; Li et al. 2018b; Milanova et al. 2002, 2005; Smaragdakis and Balatsouras 2015; Smaragdakis et al. 2011, 2014; Sridharan et al. 2013; Tan et al. 2016, 2017; Thakur and Nandivada 2019; Thiessen and Lhoták 2017].

We summarize the research work on context-sensitive pointer analysis into three categories according to (1) the kinds of context elements, (2) the composition of context elements, and (3) the selection of which parts of a given program to analyze with context sensitivity. Interestingly, the techniques in different categories are orthogonal and can thus be combined for producing more sophisticated pointer analyses.

Kinds of Context Elements. Many elements in a program, e.g., call sites, allocation sites, and types, can be considered as context elements in context sensitivity. The earliest type of context elements for Java pointer analysis inherits from the foundational approaches used both for C and for functional languages [Sharir and Pnueli 1981; Shivers 1991] where context elements are call sites. We refer to this style of context sensitivity as call-site sensitivity or call-string sensitivity [Smaragdakis and Balatsouras 2015]. Unlike C programs, which often predominantly manipulate values in the stack, Java programs have their inter-procedural data flow primarily through heap objects. Accordingly, Milanova et al. [2005] proposed object sensitivity that uses allocation sites of receiver objects (the program points where these objects are created) as context elements. Generally, object sensitivity is more precise and efficient than call-site sensitivity, and is considered the most effective context-sensitivity variant for producing good precision for Java. This conclusion has been extensively validated in various pointer analysis frameworks [Kastrinis and Smaragdakis 2013; Lhoták and Hendren 2006; Naik et al. 2006; Sridharan et al. 2013; Tan et al. 2016, 2017].

Despite being precise, object sensitivity is difficult to scale for large and complex Java programs, which motivated the concept of type sensitivity [Smaragdakis et al. 2011]. In type sensitivity, the context elements are the types of the classes containing allocation sites (as the latter would appear in object sensitivity). This more coarse-grained choice of context elements enables type sensitivity to run much faster by sacrificing some precision compared to object sensitivity. Today, call-site, object, and type sensitivity are still considered the three mainstream context-sensitivity variants for Java pointer analysis [Jeong et al. 2017; Smaragdakis and Balatsouras 2015; Tan et al. 2017].

Composition of Context Elements. With context sensitivity, each method can be analyzed in multiple contexts, where a context consists of a list of context elements that model the run-time call stack. For example, in call-string sensitivity [Sharir and Pnueli 1981; Shivers 1991], a context for a method m is composed by the context elements that are listed as $[c_1, c_2, \ldots]$, where c_1 is m's call site, c_2 is the call site of the method that contains c_1, etc. To ensure termination of the analysis, a *k-limiting* approach is typically followed: only the most recent k context elements are selected, and, for efficiency, k is usually set to a very small value in practice. The call-string sensitivity approach has been adopted in virtually all context-sensitive pointer analyses [Smaragdakis and Balatsouras 2015; Sridharan et al. 2013]. However, Tan et al. [2016] found that this conventional approach leads to many context elements that are not useful for improving precision while occupying the limited slots of context elements determined by k. To address this problem, Tan et al. [2016] presented an approach to identify such redundant context elements. By skipping those elements, a resulting

```
1 Object m(Object o){
2   return o;
3 }
4 x1 = new A();
5 x2 = m(x1);
6 y1 = new B();
7 y2 = m(y1);
```

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

context-sensitive pointer analysis is guaranteed to be at least as precise as (practically, more precise than) the conventional context sensitivity for the same choice of k. Similarly, Jeon et al. [2018] developed a machine-learning approach to select context elements, leading to a context-sensitive pointer analysis that is both reasonably efficient and precise.

Selective Use of Context Sensitivity. The conventional approach to context sensitive analysis is to uniformly apply context sensitivity to every method in the given program. However, in recent years, researchers have become aware that for some methods, context sensitivity does not help improve the analysis precision but only introduces extra analysis cost. As a result, many selective context-sensitive pointer analyses have been proposed and contribute to a promising research direction for making practical trade-offs between precision and efficiency [Hassanshahi et al. 2017; Jeon et al. 2019; Jeong et al. 2017; Li et al. 2018b; Smaragdakis et al. 2014]. In these analyses, context sensitivity is applied only for the methods where it is deemed beneficial to the overall analysis precision. However, finding such precision-critical methods is challenging. The existing approaches to selective context-sensitive pointer analyses rely on heuristics [Hassanshahi et al. 2017; Smaragdakis et al. 2014] or machine learning [Jeon et al. 2019; Jeong et al. 2017]. The heuristic approaches require manual tuning of multiple complicated parameters. The machine learning approaches are able to reveal some program features that may correlate with the analysis effectiveness; however, the weaknesses of machine learning approaches are also well known: they requires training and manual oversight during the tuning phase, they can behave unpredictably for new inputs, and they offer few insights on why they work.

In this article, we present a more principled approach that explains when using context sensitivity for a method is beneficial for precision, or conversely, when omitting context sensitivity introduces imprecision.

3 CAUSES OF IMPRECISION IN CONTEXT-INSENSITIVE POINTER ANALYSIS

To address the challenge of how to predict which methods are precision-critical in a given program, in this section, we introduce a general model to show that most of the precision loss in context-insensitive pointer analysis for Java can be expressed in terms of three patterns of value flows, or as combinations of these. 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]. We first introduce the three precision loss patterns and then describe three corresponding concrete examples (Sections 3.1–3.3). This characterization of precision loss provides the conceptual foundation for Zipper and Zipper^e to identify the methods that will be analyzed with context sensitivity, as explained in Sections 4 and 5, respectively.

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

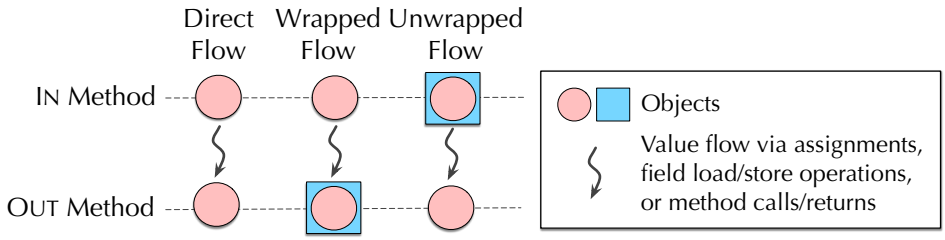

Fig. 2. Three general patterns of value flow that cause precision loss in context-insensitive analysis.

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.

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 3.1 (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 3.2 (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 3.3 (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 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 3.4 (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 3.5 (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.

Direct Flow

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

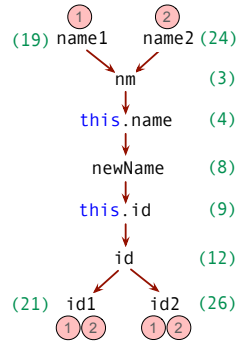

Fig. 3. Example of direct flow. (The line number for each variable/field reference on the right-hand side is shown in parentheses.)

3.1 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 → line 8 → line 9 → line 12), and finally out of the Out method `getID` to `id1` and `id2` in lines 21 and 26. Hence, by Definition 3.3, 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 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.

3.2 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

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 }
```

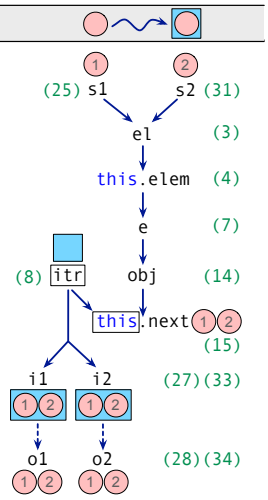

Fig. 4. Example of wrapped flow.

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, □, (created on line 8) in which object ① or ② is stored, flows out of this Out method.

Object wrapping (Definition 3.2) 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 3.4, 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

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 }
```

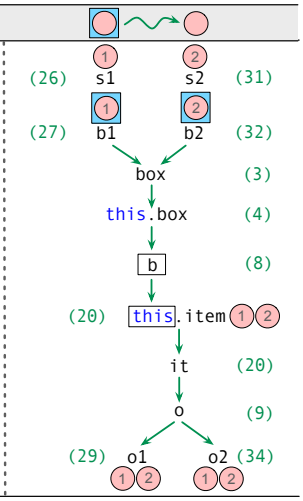

Fig. 5. Example of unwrapped flow.

the Out method must be *in the same class*, although the value flow may involve other classes, as described in Definitions 3.3—3.5. 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 4.

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 3.1, 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 3.1.

3.3 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 $\boxed{}$ 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 [1] and [2] do not flow out of the Out method `getItem` of class `SyncBox`; instead, the two unwrapped objects ① and ② (respectively stored in [1] and [2]) are the ones that flow out of this Out method.

Object unwrapping (Definition 3.2) occurs in line 20, as a result of the call in line 9: the `Box` objects ([1] and [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 3.5, the green arrows (in Figure 5) form an unwrapped flow.

We can observe that objects [1] and [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 3.2, 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`).

3.4 Combination of Flows

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 6) show that the patterns and their combinations account for essentially all the imprecision that may appear in context-insensitive analysis.

4 ZIPPER

This section introduces Zipper: our approach for identifying precision-critical methods based on the precision loss patterns of Section 3. 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 4.1 and the concepts of *object flow graphs* and *precision flow graphs* in Sections 4.2 and 4.3, respectively.

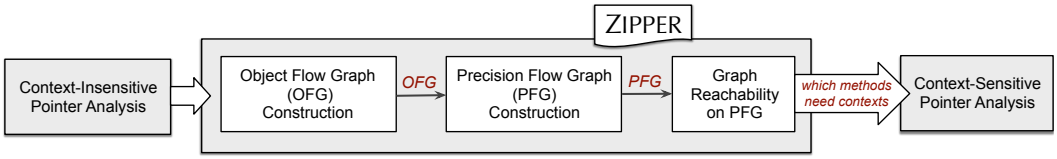

Fig. 6. Overview of Zipper.

4.1 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 4.2. 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.[1] 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 map $pt(\mathsf{v})$ 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 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.

4.2 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 4.1: 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 $\mathsf{a} \rightarrow \mathsf{b}$ in the OFG means that the objects pointed by pointer a may flow to (and also be pointed to by) pointer b.

[1] As part of a two-phase analysis, the pre-analysis of Zipper needs to be fast; so currently we use a context-insensitive pointer analysis as pre-analysis. It would be interesting future work to further explore the effect of more precise pre-analysis.

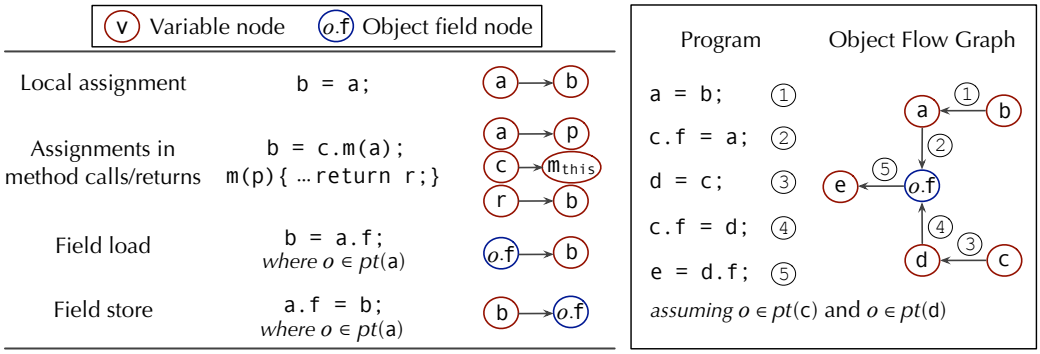

Fig. 7. Object flow graph construction, with an example.

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 3. 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 $pt(\mathsf{v})$ 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.

4.3 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 4.2, 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 3, we construct one PFG for each class in the program. As the OFG can already describe direct flow (Section 4.2), 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

Algorithm 1: PFGBUILDER

```
                OFG    (Object Flow Graph)
Input  :        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_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 ∈ VisitedNodes then
8          return
9      add N to VisitedNodes
10     if N is a variable node N_a then
11         foreach [b = a.f] ∈ S do                    // Handling unwrapped flow
12             add N_a → N_b to WUEdges
13         foreach [b.f = a] ∈ S do                    // Handling wrapped flow
14             foreach o ∈ pt(b) do
15                 add N_a → N_[o] to WUEdges
16     foreach N → N' ∈ OFG ∪ WUEdges do
17         add N → N' to PFG_c
18         Dfs(N')
```

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 3), 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 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_{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_{a} to node N_{b}. This allows us to model unwrapped flow, as defined in Definition 3.5 and illustrated in Section 3.3.

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_{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

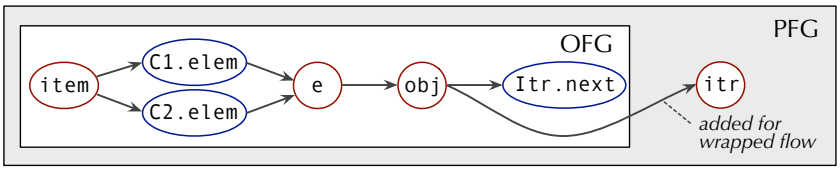

Fig. 8. A partial PFG for class `Collection` in Figure 4 (wrapped flow). `C1`, `C2`, and `Itr` denote the objects of classes `Collection` and `Iterator` allocated in lines 24, 30, and 8 in Figure 4, respectively.

node N_{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 3.4 and illustrated in Section 3.2. As an example, for the object wrapping `this.next = obj` of line 15 in Figure 4, $pt(\mathtt{this})$ 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_{a} to $N_{[\mathtt{o}]}$ we had added an edge from N_{a} to N_{b} (mirroring the handling of unwrapped flows), we would miss some flows. Conceptually, according to Definition 3.4, 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 3.4.

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_{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 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

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* $\leftarrow \{\}$, $\mathsf{PCM}_c \leftarrow \{\}$
2 **foreach** $m \in \textsc{Out}_c$ **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

Algorithm 3: Zipper

Input : p (Input program)
OFG (Object Flow Graph for program p)
Output: PCM (Precision-Critical Methods for program p)

1 PCM $\leftarrow \{\}$, $S \leftarrow$ set of all statements in p
2 **foreach** *class c in p* **do**
3 $\mathsf{PFG}_c \leftarrow$ PfgBuilder(OFG, c, S)
4 $\mathsf{PCM}_c \leftarrow$ PcmCollector(c, PFG_c)
5 PCM $\bigcup$= PCM_c
6 **return** PCM

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 $\textsc{Out}_c$) 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.

Algorithm 3 shows how Zipper uses Algorithms 1 and 2 to identify the precision-critical methods PCM for a given program p. 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 article we only consider context sensitivity at the granularity of methods, and leave the potential of more refined options for future work.

5 ZIPPER EXPRESS

In Section 3, we have explained where precision is lost in a context-insensitive pointer analysis in terms of the three patterns of precision-loss flows. Based on this principle, Zipper (Section 4) has been designed to apply context sensitivity only to the identified precision-critical methods, while analyzing the remaining methods context-insensitively. Zipper fulfills its goal (as demonstrated in Section 6.1): it is able to preserve virtually all of the precision of highly-precise conventional context-sensitive pointer analyses while making them run faster. One can use Zipper as a full replacement for a conventional context-sensitive pointer analysis such as `2obj`.

However, Zipper's precision-preserving principle is very strict in the sense that it will not sacrifice any precision for improving efficiency. On one hand, this design choice is suitable for validating whether our precision-loss model (Section 3) can help identify thoroughly the precision-critical methods that capture the vast majority of precision of full context sensitivity; on the other hand, it may make Zipper unscalable for some complex programs.

To address this problem, in this section, we present a new pointer analysis, Zipper *express* (Zipper^e). Zipper^e makes a new precision and efficiency trade-off, running significantly faster while being only slightly less precise than Zipper. Consequently, Zipper^e is a new sweet spot in state-of-the-art pointer analysis for Java, with great efficiency and good precision.

5.1 Insights of Zipper^e

Among the precision-critical methods identified by Zipper, some may significantly degrade analysis efficiency if treated context sensitively. Therefore, to make the analysis run fast, we need to find those *efficiency-critical methods*. The idea behind Zipper^e is to apply context sensitivity only to the precision-critical methods that are not efficiency-critical, i.e., those that do not significantly hurt analysis efficiency.

How to Identify Efficiency-Critical Methods? In Zipper^e, we consider the size of the points-to set for each method m, denoted as $\#\texttt{pts}_m$, as an important factor for determining whether a method may incur serious efficiency problems when being analyzed context sensitively. Here the size of the points-to set for a method m means the sum of the sizes of the points-to sets for all the variables in m. As the analysis is built on top of the three-address code of a program, an extra variable is introduced when a parameter or field in the program is referenced in this code format. Therefore, the variables in a method include all original local variables, as well as those extra variables.

Leveraging $\#\texttt{pts}_m$ to estimate the efficiency cost for pointer analysis has been adopted in existing literature [Smaragdakis et al. 2014], and there are two reasons to consider this metric. First, the efficiency of a context-sensitive pointer analysis is directly correlated with the number of its generated points-to facts: the larger the number, the more memory and analysis iterations are needed, and, accordingly, more analysis time is spent. Second, $\#\texttt{pts}_m$ can easily be approximated

from a context-insensitive pre-analysis as described in Section 4. If $\#\texttt{pts}_m$ for a method m is large, it indicates that analyzing m context-sensitively would likely produce many context-sensitive points-to facts [Li et al. 2018b] resulting in a high analysis time.

However, we find that simply using $\#\texttt{pts}_m$ to decide whether a given method m is efficiency-critical does not work well: the analysis becomes both inefficient and imprecise for some programs as the wrong methods are selected for context sensitivity. At the same time, we also do not want to complicate Zipper^e by incorporating multiple configurable parameters as in existing work [Hassanshahi et al. 2017; Smaragdakis et al. 2014; Wei and Ryder 2015].

The precision-loss model introduced in Section 3 enables us to address this problem. Recall that the precision-critical methods are identified based on the granularity of each class (from its In methods to its Out methods). This means that, for each class, if one method that is involved in the precision-loss flows (or their combinations) is not analyzed with context sensitivity, we may still lose the precision even if all the remaining involved methods are analyzed context-sensitively.

Based on this key observation, in Zipper^e we identify efficiency-critical methods using $\#\texttt{pts}_m$ but at the granularity of classes rather than methods. This means that we use the sum of $\#\texttt{pts}_m$ of all the methods m that are involved in the precision-loss flows (or their combinations) when analyzing class c, and we denote this parameter as $\#\texttt{pts}_c$ (note that m may belong to a class other than c, as explained in Section 3). For each class c, all its involved methods can be treated as one: either *all* or *none* of its methods are analyzed with context sensitivity.

How to Identify Efficiency-Critical Methods Using $\#\texttt{pts}_c$*?* When $\#\texttt{pts}_c$ for a class c is large, i.e., exceeds a certain threshold, Zipper^e excludes all its related methods that are involved in the precision-loss flows from context sensitivity, thereby preventing them from hurting the analysis speed. Note that, as explained above, $\#\texttt{pts}_c$ is not the points-to size for c; instead, it represents the cumulative points-to size of all precision-critical methods involved in the precision-loss flows starting from and ending in class c.

The last problem is how to select a shared threshold for different programs. A fixed value will not work well as a constant threshold may be too large for small or simple programs, or too small for large or complex programs. To resolve this problem, instead of a fixed value, we consider the threshold as a percentage value, PV. That is, for each class c in program p, if $(\#\texttt{pts}_c \,/\, \#\texttt{pts}_p) > \text{PV}$ (where $\#\texttt{pts}_p$ is the sum of the sizes of the points-to sets for all variables in p), we say that $\#\texttt{pts}_c$ is too large relative to the points-to size of the whole program, and thus consider the related methods in c to be efficiency-critical methods. With this heuristic, we only need *one* threshold, i.e., PV, for *all* programs. By default PV is set to 5%.

5.2 The Zipper^e Algorithm

Given the background and insights presented in Section 5.1, in this section we explain how Zipper^e works. Algorithm 4 describes how Zipper^e selects a set of methods, i.e., M_{cs} (the output of the algorithm), that will be analyzed context-sensitively in the main analysis. Briefly, they are the remaining methods after excluding the efficiency-critical methods from the precision-critical methods identified by the approach as used in Zipper. Thus Algorithm 4 extends Algorithm 3.

As inputs of the algorithm, OFG is the object flow graph, which is built by the pre-analysis as described in Section 4.2, and PV is the shared threshold for all programs as described in Section 5.1.

In line 1, *TH* is per program and denotes the threshold for the given program p, representing the efficiency-critical bound for the sizes of the points-to set of each class c in p. As illustrated in Section 5.1, when $\#\texttt{pts}_c$ is larger than *TH*, Zipper^e will consider all precision-critical methods related to class c to be efficiency-critical. As shown in line 6, $\#\texttt{pts}_c$ is the size of the points-to sets

Algorithm 4: ZIPPER^e

Input : p (Input program)
OFG (Object Flow Graph for program p)
PV (The shared threshold for all programs)
Output: M_{cs} (The methods to be analyzed context-sensitively)

1 $TH = PV \times \sum_{v \in p} |pt(v)|$ `// The threshold for program` p
2 $\mathsf{M}_{cs} \leftarrow \{\}$, $S \leftarrow$ set of all statements in p
3 **foreach** *class c in p* **do**
4 $\quad \mathsf{PFG}_c \leftarrow$ PFGBUILDER(OFG, c, S)
5 $\quad \mathsf{PCM}_c \leftarrow$ PCMCOLLECTOR(c, PFG_c)
6 $\quad \#\mathtt{pts}_c = \sum_{m \in \mathsf{PCM}_c} \sum_{v \in m} |pt(v)|$
7 $\quad$ **if** $\#\mathtt{pts}_c \leq TH$ **then**
8 $\quad\quad \mathsf{M}_{cs} \bigcup= \mathsf{PCM}_c$
9 **return** M_{cs}

for all the variables in all precision-critical methods for each given class c. Per lines 7–8, only the precision-critical methods that are not efficiency-critical will be analyzed with context sensitivity.

As explained in Section 4, it is possible that a method m is involved in precision-loss flows for multiple classes, say c_1 and c_2, when collecting their corresponding precision-critical methods. In ZIPPER^e, if m is identified as efficiency-critical in c_1 but non-efficiency-critical in c_2, m will still be analyzed context-sensitively for precision, as reflected in the set union operation of line 8.

6 EVALUATION

In this section, we investigate the following research questions for evaluating our selective context-sensitivity techniques.

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?

We consider object-sensitive pointer analysis [Milanova et al. 2002, 2005] as the conventional analysis that applies context sensitivity to all methods. That analysis is well-recognized as the most precise context-sensitivity variant and widely supported by all popular pointer analysis frameworks for Java.

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?

RQ4. Is the precision-loss model (introduced in Section 3) general enough to preserve the precision of other mainstream variants of context sensitivity, i.e., call-site sensitivity [Sharir and Pnueli 1981], type sensitivity [Smaragdakis et al. 2011], and hybrid context sensitivity [Kastrinis and Smaragdakis 2013]?

RQ5. Is ZIPPER^e-guided pointer analysis effective in practice?
- (a) How does ZIPPER^e-guided pointer analysis perform in terms of precision and efficiency?

(b) What is the overhead of running Zipper^e and how many methods does Zipper^e consider precision-critical and not efficiency-critical?

Implementation. We have implemented Zipper and Zipper^e as open-source stand-alone tools in Java, available at http://www.brics.dk/zipper. Benefiting from simple insights and algorithms, Zipper's core implementation contains less than 1 500 lines of Java code, and Zipper^e adds only about 100 lines of Java code on top of Zipper. In addition, both tools are 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 and Zipper^e 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 (or Zipper^e's) output is just a set of precision-critical methods. 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 or Zipper^e. 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 (with 16 cores[2]) 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.

As explained in Section 5, Zipper^e differs from Zipper by being designed to make a context-sensitive pointer analysis run very fast with still good precision. Therefore, to evaluate the effectiveness of Zipper^e, in addition to the 10 programs in Table 1, we consider another 10 large programs (the last 10 entries in Table 8) where both `2obj` and Zipper-guided object-sensitive analysis (Zipper-2obj) cannot finish the analysis within 3 hours. They are `jython` (interpreter for Python), `hsqldb` (database system), `pmd5` (source code analyzer), `jedit` (text editor), `soot` (analysis framework for Java), `eclipse-r` (reflection-enabled eclipse), `briss` (PDF cropper), `h2` (database system), `gruntspud` (graphical CVS client) and `columba` (email client). As mentioned in Section 5, we set PV to its default value 5% in the experiments.

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

[2]The implementation of Zipper and Zipper^e trivially exploits multi-threading, whereas all the pointer analyses run single-threaded.

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.

Other DaCapo benchmarks. In the DaCapo benchmarks, `2obj` fails to scale for `jython` and `hsqldb` within 1.5 hours. Zipper also cannot help for these two known problematic benchmarks [Kastrinis and Smaragdakis 2013; Smaragdakis et al. 2011; Tan et al. 2016, 2017], as, unlike Zipper^e and 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`), for Zipper, we only consider the remaining five large DaCapo benchmarks for which `2obj` is scalable. For Zipper^e, `jython` and `hsqldb` are also included, as mentioned above.

6.1 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 6.2.

6.1.1 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.

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. 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 3), most context-related imprecision can be discovered by Zipper. However, on

Table 1. Performance and precision results 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	84	2 961	4 681	19 197	101 616
	2obj	3 300	1 606	3 491	16 859	76 807
	Zipper-2obj	1 037	1 614	3 501	16 863	76 858
	introA-2obj	265	2 675	4 262	19 011	97 120
	introB-2obj	2 527	2 149	3 997	18 703	90 126
checkstyle	ci	51	1 114	1 444	9 866	57 490
	2obj	2 003	581	1 035	9 513	48 809
	Zipper-2obj	378	607	1 059	9 526	48 945
	introA-2obj	134	970	1 206	9 769	55 736
	introB-2obj	1 781	792	1 134	9 595	51 437
sunflow	ci	62	3 003	4 113	19 773	106 410
	2obj	1 208	1 837	3 385	19 245	89 866
	Zipper-2obj	567	1 869	3 391	19 247	89 902
	introA-2obj	160	2 764	3 796	19 651	103 536
	introB-2obj	413	2 346	3 529	19 429	95 602
findbugs	ci	53	2 508	2 925	13 036	77 370
	2obj	2 661	1 409	2 182	12 657	65 836
	Zipper-2obj	908	1 437	2 190	12 662	65 880
	introA-2obj	196	2 271	2 422	12 960	73 681
	introB-2obj	419	2 024	2 372	12 882	70 725
jpc	ci	57	2 370	5 013	17 146	96 669
	2obj	559	1 392	4 222	15 852	81 030
	Zipper-2obj	229	1 415	4 231	15 857	81 072
	introA-2obj	132	2 169	4 703	17 038	95 170
	introB-2obj	329	1 736	4 327	16 001	85 316
eclipse	ci	26	1 139	1 334	8 465	45 474
	2obj	146	546	980	7 911	38 151
	Zipper-2obj	72	586	1 013	7 927	38 369
	introA-2obj	59	977	1 118	8 319	43 781
	introB-2obj	75	764	1 046	8 001	39 876
chart	ci	50	1 810	1 852	12 064	63 453
	2obj	282	883	1 378	11 330	52 374
	Zipper-2obj	82	910	1 384	11 334	52 399
	introA-2obj	135	1 580	1 613	11 952	61 323
	introB-2obj	198	1 236	1 497	11 518	55 594
fop	ci	78	2 458	3 585	17 154	84 330
	2obj	1 200	1 446	2 844	16 438	71 408
	Zipper-2obj	520	1 471	2 860	16 442	71 478
	introA-2obj	212	2 206	3 246	17 007	82 113
	introB-2obj	561	1 804	2 979	16 571	75 770
xalan	ci	43	1 182	1 898	9 705	51 302
	2obj	1 093	533	1 522	9 047	44 871
	Zipper-2obj	116	568	1 542	9 129	45 332
	introA-2obj	117	1 129	1 765	9 637	50 659
	introB-2obj	755	723	1 579	9 119	45 904
bloat	ci	32	1 924	2 014	8 939	61 150
	2obj	3 525	1 193	1 427	8 470	53 143
	Zipper-2obj	2 971	1 224	1 449	8 486	53 289
	introA-2obj	61	1 809	1 690	8 869	60 111
	introB-2obj	141	1 621	1 522	8 626	55 455
	Zipper-2obj*	53	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.

```
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.

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, we give 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.

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 method `m`. This case is rare, since it is unusual in Java (or, generally, object-oriented) 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.

6.1.2 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 3.4× speedup compared with `2obj`. The best case is program `xalan` where `2obj` spends about 18 minutes while Zipper-guided analysis finishes running in well under 2 minutes (9.4× speedup). The worst case is program `bloat` where `2obj` spends 59 minutes while Zipper-guided analysis is 9 minutes faster (1.2× 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. Specifically, 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 66.5×, which is much faster than the original 1.2×; 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. A promising approach in this direction is Zipper^e, as already discussed. Comparing with the Zipper^e numbers in Section 6.6, it can be seen that Zipper^e outperforms the modified Zipper (Zipper-2obj*), in both efficiency and precision, for the very same benchmark (bloat) that motivated Zipper-2obj* in the first place.

Finally, note that, as Zipper only reports on average 38% of the methods in a program as precision-critical (see Section 6.3.1), 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.

6.1.3 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 54 seconds for each program.

Table 2 (last row) shows the performance of Zipper itself: the average analysis time of Zipper is just 11 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 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.

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

Program	#classes	#nodes in OFG	#edges in OFG	#avg. nodes in PFG	#avg. edges in PFG	Zipper time (seconds)
batik	2 701	189 993	486 629	3 576	10 253	27
checkstyle	1 301	92 167	203 445	3 333	8 383	7
sunflow	2 496	188 395	407 526	3 080	8 057	17
findbugs	1 752	127 939	276 319	1 711	4 273	6
jpc	2 039	166 639	386 618	2 329	6 727	15
eclipse	1 122	84 352	172 161	1 823	4 480	3
chart	1 578	115 515	230 409	2 464	6 086	8
fop	2 580	162 086	370 003	2 285	6 403	17
xalan	1 268	94 969	207 300	3 061	7 478	5
bloat	1 107	87 223	201 057	1 942	4 724	4
avg.	1 794	130 928	294 147	2 560	6 686	11

6.2 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 analyess 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 11 seconds (as shown in Section 6.1.3).

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 as no surprise given that the precision of IntroA is only slightly better than context-insensitive analysis while Zipper preserves almost all of the precision of a full context-sensitive analysis, 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.

6.3 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 3. 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).

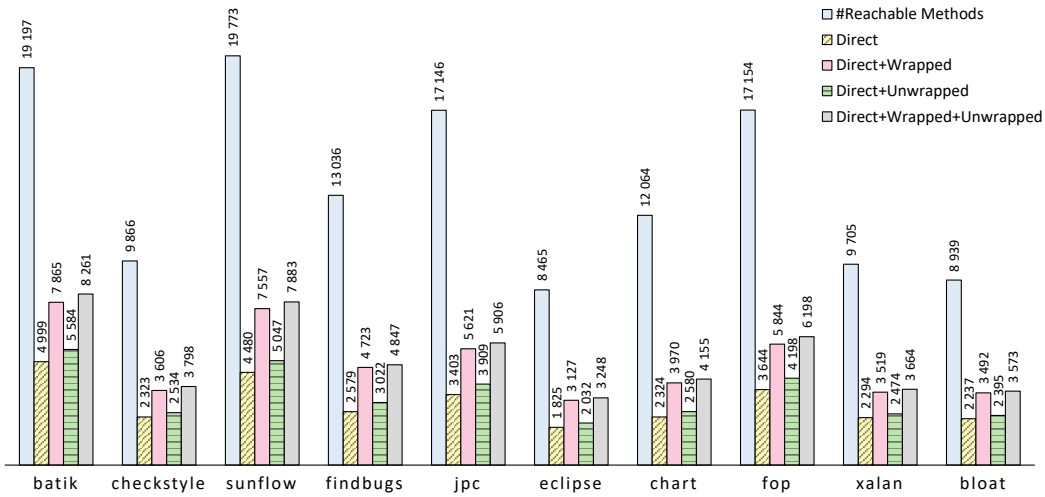

Fig. 11. Precision-critical methods under different combinations of the basic precision loss patterns.

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.

We first evaluate the number of precision-critical methods reported by Zipper under different flow combinations in Section 6.3.1, and then present the precision and efficiency of Zipper-guided pointer analyses with respect to the different flow combinations in Section 6.3.2.

6.3.1 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 6.1.1, 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 24.8% under `Direct+Unwrapped`, which shows that wrapped flow introduces significantly more precision-critical methods than unwrapped flow. `Direct+Unwrapped` introduces 2.5% 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.

6.3.2 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

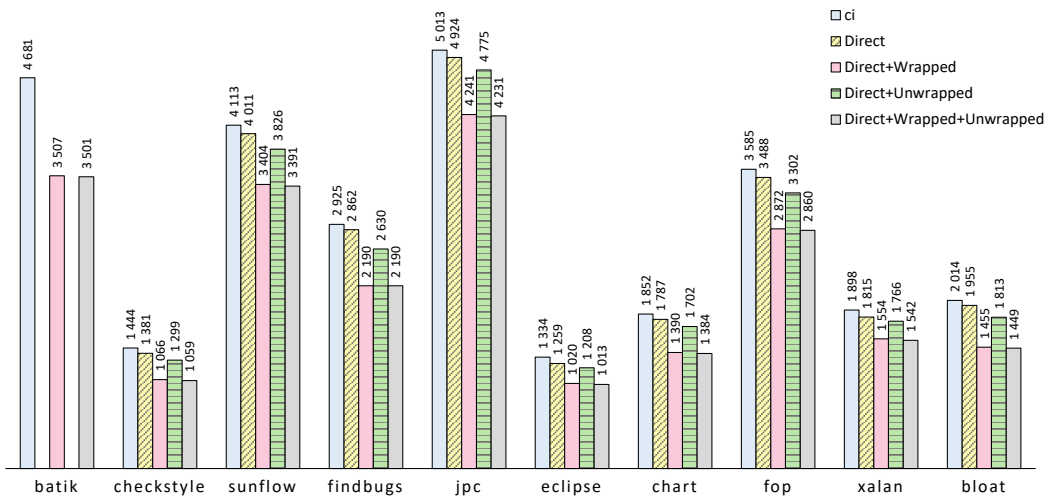

Fig. 12. #poly-call for different combinations of the basic precision loss patterns.

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.

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 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 improves 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 185 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 6.3.1.

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 runs 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`.

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

	batik	checkstyle	sunflow	findbugs	jpc	eclipse	chart	fop	xalan	bloat
Direct	–	529	1 690	2 486	2 100	128	441	3 020	283	3 254
Direct+Wrapped	926	302	556	932	230	75	83	471	127	3 024
Direct+Unwrapped	–	513	3 039	3 255	3 630	117	633	5 664	278	3 439
Direct+Wrapped+Unwrapped	1 037	378	567	908	229	72	82	520	116	2 971

Table 4. Precision metrics of different analyses on the simple 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	luindex	ci	734	940	6 670	33 130
	2obj	428	1 520	7 357	49 348		2obj	297	675	6 256	29 021
	Zipper-2obj	452	1 530	7 361	49 400		Zipper-2obj	327	686	6 259	29 076
	introA-2obj	990	1 694	7 783	53 071		introA-2obj	617	802	6 600	32 370
	introB-2obj	640	1 560	7 448	50 257		introB-2obj	450	714	6 316	29 835
lusearch	ci	844	1 133	7 352	36 343	pmd	ci	1 263	1 039	8 427	42 415
	2obj	299	850	6 904	31 811		2obj	657	718	7 648	35 563
	Zipper-2obj	322	864	6 907	31 869		Zipper-2obj	676	728	7 654	35 626
	introA-2obj	681	981	7 277	35 531		introA-2obj	1 136	882	8 351	41 674
	introB-2obj	462	891	6 970	32 656		introB-2obj	859	777	7 929	37 379

These results clearly demonstrate that losing precision may also introduce performance decline. This is especially common for context-sensitive pointer analysis, as spurious data flow (caused by imprecision) will be replicated and propagated under different contexts, which can make the 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 1 037 seconds.

6.4 Precision of Zipper for Simple Programs

We also evaluate Zipper's precision for those simple DaCapo 2006 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 quickly), 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 4 shows the precision results of Zipper for the four simple DaCapo 2006 benchmarks. The results demonstrate that Zipper is able to preserve most of the precision (98.3% on average) of `2obj` even for such programs, which are outside the target domain of Zipper.

6.5 RQ4: Zipper for Other Context-Sensitivity Variants

We have demonstrated that Zipper is able to effectively preserve the precision of object-sensitivity, which is the most precise conventional context-sensitivity variant for Java (Section 6.1). However, it is beneficial to further investigate whether Zipper (and its underlying precision-loss principle) is robust enough to preserve most of the precision also for other context-sensitivity variants.

Table 5. Performance and precision results for conventional call-site-sensitive (2cs) and Zipper-guided (Zipper-2cs) pointer analyses.

Program	Pointer analysis	Time (s)	#fail-cast	#poly-call	#reach-mtd	#call-edge
batik	2cs	6 886	2 452	4 281	18 882	94 211
	Zipper-2cs	2 534	2 453	4 291	18 883	94 240
checkstyle	2cs	2 277	863	1 285	9 766	54 171
	Zipper-2cs	147	869	1 306	9 771	54 268
sunflow	2cs	5 570	2 504	3 827	19 556	100 701
	Zipper-2cs	1 232	2 515	3 831	19 557	100 722
findbugs	2cs	3 812	2 056	2 648	12 926	72 118
	Zipper-2cs	514	2 069	2 657	12 927	72 146
jpc	2cs	3 343	1 855	4 616	16 350	89 677
	Zipper-2cs	813	1 857	4 621	16 352	89 704
eclipse	2cs	1 896	886	1 186	8 195	42 872
	Zipper-2cs	144	887	1 198	8 198	42 923
chart	2cs	2 705	1 481	1 691	11 722	59 691
	Zipper-2cs	552	1 483	1 696	11 725	59 707
fop	2cs	5 503	1 975	3 254	16 700	79 524
	Zipper-2cs	1 838	1 977	3 264	16 700	79 578
xalan	2cs	1 927	919	1 738	9 339	48 763
	Zipper-2cs	227	921	1 770	9 342	49 031
bloat	2cs	5 712	1 699	1 793	8 703	58 696
	Zipper-2cs	3 308	1 701	1 816	8 717	58 819

In this section, we consider three representative context-sensitivity variants: 2-call-site-sensitivity (`2cs`) for call-site sensitivity [Sharir and Pnueli 1981], 2-type-sensitivity (`2type`) for type sensitivity [Smaragdakis et al. 2011] and selective-2-object-sensitivity (`s2obj`) for hybrid context sensitivity [Kastrinis and Smaragdakis 2013]. These variants often appear in recent literature [Jeon et al. 2018; Jeong et al. 2017; Smaragdakis et al. 2014; Tan et al. 2016, 2017]. The precision and performance results of the three analyses and their corresponding Zipper-guided versions are shown in Tables 5–7. On average, Zipper-guided pointer analysis achieves 6.6×, 1.3×, and 2.9× speedups for `2cs`, `2type`, and `s2obj`, respectively. The precision results are summarized as follows.

- For call-site sensitivity, on average 99.7% of the precision of `2cs` can be preserved by Zipper considering all client analyses. Specifically, the average number for each client analysis is 99.7% for #fail-cast, 99.3% for #poly-call, 99.97% for #reach-mtd and 99.9% for #call-edge.
- For type sensitivity, on average 99.2% of the precision of `2type` can be preserved by Zipper considering all client analyses. Specifically, the average number for each client analysis is 97.9% for #fail-cast, 99.2% for #poly-call, 99.9% for #reach-mtd and 99.8% for #call-edge.
- For hybrid (call-site + object) sensitivity, on average 99.0% of the precision of `s2obj` can be preserved by Zipper considering all client analyses. Specifically, the average number for each client analysis is 97.6% for #fail-cast, 99.0% for #poly-call, 99.8% for #reach-mtd and 99.7% for #call-edge.

In summary, these results demonstrate that the theoretical foundation of Zipper is general enough to effectively identify the precision-critical methods for a wide range of conventional context sensitivity variants.

Table 6. Performance and precision results for conventional type-sensitive (2type) and Zipper-guided (Zipper-2type) pointer analyses.

Program	Pointer analysis	Time (s)	#fail-cast	#poly-call	#reach-mtd	#call-edge
batik	2type	378	1 938	3 623	16 892	77 337
	Zipper-2type	239	1 941	3 617	16 894	77 351
checkstyle	2type	125	695	1 122	9 534	49 274
	Zipper-2type	82	711	1 140	9 544	49 436
sunflow	2type	197	2 247	3 506	19 315	90 967
	Zipper-2type	136	2 262	3 510	19 316	91 022
findbugs	2type	265	1 683	2 345	12 674	66 443
	Zipper-2type	179	1 703	2 349	12 678	66 488
jpc	2type	128	1 599	4 328	15 908	81 527
	Zipper-2type	98	1 614	4 336	15 911	81 559
eclipse	2type	57	665	1 031	7 933	38 337
	Zipper-2type	50	714	1 063	7 967	38 677
chart	2type	84	1 155	1 446	11 439	52 965
	Zipper-2type	79	1 175	1 451	11 444	53 011
fop	2type	251	1 753	2 930	16 477	71847
	Zipper-2type	189	1 777	2 943	16 482	71 922
xalan	2type	99	729	1 565	9 151	45 444
	Zipper-2type	79	758	1 578	9 160	45 566
bloat	2type	74	1 486	1 626	8 523	54 279
	Zipper-2type	73	1 508	1 642	8 536	54 424

Table 7. Performance and precision results for conventional hybrid context-sensitive (s2obj) and Zipper-guided (Zipper-s2obj) pointer analyses.

Program	Pointer analysis	Time (s)	#fail-cast	#poly-call	#reach-mtd	#call-edge
batik	s2obj	3 619	1 463	3 485	16 857	76 751
	Zipper-s2obj	1 040	1 471	3 493	16 861	76 795
checkstyle	s2obj	1 226	500	1 030	9 511	48 794
	Zipper-s2obj	280	516	1 053	9 522	48 936
sunflow	s2obj	1 466	1 674	3 376	19 243	89 831
	Zipper-s2obj	612	1 700	3 380	19 245	89 862
findbugs	s2obj	2 485	1 326	2 181	12 656	65 835
	Zipper-s2obj	941	1 346	2 189	12 661	65 875
jpc	s2obj	672	1 240	4 217	15 850	80 996
	Zipper-s2obj	257	1 251	4 223	15 855	81 031
eclipse	s2obj	121	456	979	7 910	38 149
	Zipper-s2obj	68	480	1 012	7 926	38 361
chart	s2obj	301	757	1 378	11 328	52 370
	Zipper-s2obj	105	774	1 383	11 332	52 392
fop	s2obj	1 290	1 295	2 834	16 436	71 370
	Zipper-s2obj	528	1 308	2 846	16 439	71 432
xalan	s2obj	670	447	1 518	9 043	44 787
	Zipper-s2obj	118	473	1 541	9 128	45 328
bloat	s2obj	3 485	1 125	1 426	8 469	53 142
	Zipper-s2obj	3 011	1 146	1 447	8 485	53 287

6.6 RQ5: Effectiveness of Zipper^e-Guided Pointer Analysis

Zipper^e offers the promise of being almost as precise as the default Zipper algorithm, yet with significantly enhanced scalability. We evaluate the effectiveness of Zipper^e-guided pointer analysis using 20 input programs, including 10 large or complex programs for which both `2obj` and Zipper-guided object-sensitive analysis (Zipper-2obj) are not scalable. To the best of our knowledge, compared with existing literature about whole-program pointer analysis for Java, this experiment includes the largest set of hard-to-analyze Java programs. All 20 programs and the analysis results for them are shown in Table 8.

6.6.1 Efficiency and Precision of Zipper^e-Guided Pointer Analysis. Generally, on the 10 programs for which `2obj` scales, Zipper-guided pointer analysis (Zipper^e-2obj) preserves 94.7% of the precision for `2obj` for a speedup of 25.5×, on average. In addition, on the 10 programs for which `2obj` fails to scale (i.e., does not terminate in 3 hours), on average, Zipper^e can guide `2obj` to finish analyzing them in less than 11 minutes with good precision.

To better understand the efficiency and precision trade-off made by Zipper^e, let us first recall the effectiveness of other analyses in Table 8: context-insensitive pointer analysis (CI), introspective analyses, IntroA and IntroB [Smaragdakis et al. 2014]. CI is the fastest pointer analysis but with the worst precision. Being a selective context-sensitive pointer analysis, IntroA is an extremely fast and scalable pointer analysis and can achieve strictly better precision than CI, and IntroB is strictly more precise but less efficient than IntroA in practice. In addition, we compare Zipper^e with Scaler, a state-of-the-art scalability-first pointer analysis [Li et al. 2018b], which also emphasizes scalability with good precision. With this background information, we summarize the results in Table 8 as follows.

- Zipper^e-2obj is scalable for 19 programs, and IntroA is scalable for all 20 programs.
- Zipper^e-2obj is faster than IntroA in 15 out of 20 programs.
- Zipper^e-2obj is even faster than CI for 5 programs.
- Zipper^e-2obj is as fast as CI (the difference is 10% on average, ranging from 2 to 11 seconds only) for another 6 programs. IntroA is notably slower than CI (at least 2× slowdown) for most (15 out of 20) programs.
- Zipper^e-2obj is significantly more precise than IntroA in *all* 76 available precision numbers (76 = 19 programs × 4 precision metrics).
- Zipper^e-2obj is even more precise than IntroB in most cases (48 out of 60 precision numbers) on the programs for which both analyses are scalable (60 = 15 programs × 4 precision metrics, IntroB is unscalable for 4 programs).
- For the first 10 programs (for which `2obj` is scalable), on average, Scaler preserves 96.7% of the precision for `2obj` for a speedup of 4.0×, while Zipper^e-2obj preserves 94.7% of the precision for `2obj` for a speedup of 25.5×.
- For the last 10 programs (for which `2obj` is not scalable), on average, Zipper^e-2obj greatly outperforms Scaler in 7-of-10 cases (with a median speedup of 6.4×). Zipper^e-2obj and Scaler both greatly improve on the precision of CI (client precision metrics see an average 15.8% and 16.1% reduction for Zipper^e-2obj and Scaler, respectively).

Below, we further explain and discuss some interesting results.

How to Make Zipper^e-2obj Scalable for `Soot`*?* Given the results in Table 8, Zipper^e-2obj cannot finish analyzing `soot` within 3 hours—the only such case in our benchmark set. This means that some efficiency-critical methods in `soot` are still not identified and excluded by Zipper^e under its default threshold setting (PV = 5%). But this is not the end of story for `soot`. As explained in Section 5, we can tune the analysis efficiency and precision by changing the PV value. Recall that

Table 8. Performance and precision results for context-insensitive (ci), conventional object-sensitive (2obj), Zipper^e-guided (Zipper^e-2obj), introspective (introX-2obj), and Scaler-guided (Scaler) pointer analyses.

Program	Pointer analysis	Time (s)	#fail-cast	#poly-call	#reach-mtd	#call-edge
batik	ci	84	2 961	4 681	19 197	101 616
	2obj	3 300	1 606	3 491	16 859	76 807
	Zipper^e-2obj	88	1 745	3 664	16 909	77 874
	introA-2obj	265	2 675	4 262	19 011	97 120
	introB-2obj	2 527	2 149	3 997	18 703	90 126
	Scaler	544	1 913	3 597	16 931	77 517
checkstyle	ci	51	1 114	1 444	9 866	57 490
	2obj	2 003	581	1 035	9 513	48 809
	Zipper^e-2obj	87	757	1 134	9 652	50 376
	introA-2obj	134	970	1 206	9 769	55 736
	introB-2obj	1 781	792	1 134	9 595	51 437
	Scaler	260	625	1 038	9 514	49 053
sunflow	ci	62	3 003	4 113	19 773	106 410
	2obj	1 208	1 837	3 385	19 245	89 866
	Zipper^e-2obj	64	2 031	3 561	19 319	90 723
	introA-2obj	160	2 764	3 796	19 651	103 536
	introB-2obj	413	2 346	3 529	19 429	95 602
	Scaler	783	2 098	3 429	19 297	90 281
findbugs	ci	53	2 508	2 925	13 036	77 370
	2obj	2 661	1 409	2 182	12 657	65 836
	Zipper^e-2obj	97	1 488	2 215	12 689	66 274
	introA-2obj	196	2 271	2 422	12 960	73 681
	introB-2obj	419	2 024	2 372	12 882	70 725
	Scaler	292	1 452	2 195	12 676	66 177
jpc	ci	57	2 370	5 013	17 146	96 669
	2obj	559	1 392	4 222	15 852	81 030
	Zipper^e-2obj	51	1 516	4 350	15 895	81 743
	introA-2obj	132	2 169	4 703	17 038	95 170
	introB-2obj	329	1 736	4 327	16 001	85 316
	Scaler	357	1 617	4 297	16 082	81 962
eclipse	ci	26	1 139	1 334	8 465	45 474
	2obj	146	546	980	7 911	38 151
	Zipper^e-2obj	28	664	1 075	7 981	38 984
	introA-2obj	59	977	1 118	8 319	43 781
	introB-2obj	75	764	1 046	8 001	39 876
	Scaler	142	554	981	7 912	38 153
chart	ci	50	1 810	1 852	12 064	63 453
	2obj	282	883	1 378	11 330	52 374
	Zipper^e-2obj	44	1 025	1 453	11 383	52 866
	introA-2obj	135	1 580	1 613	11 952	61 323
	introB-2obj	198	1 236	1 497	11 518	55 594
	Scaler	273	976	1 402	11 530	53 198
fop	ci	78	2 458	3 585	17 154	84 330
	2obj	1 200	1 446	2 844	16 438	71 408
	Zipper^e-2obj	89	1 588	2 980	16 505	72 311
	introA-2obj	212	2 206	3 246	17 007	82 113
	introB-2obj	561	1 804	2 979	16 571	75 770
	Scaler	503	1 732	2 945	16 676	72 556
xalan	ci	43	1 182	1 898	9 705	51 302
	2obj	1 093	533	1 522	9 047	44 871
	Zipper^e-2obj	45	633	1 612	9 171	45 886
	introA-2obj	117	1 129	1 765	9 637	50 659
	introB-2obj	755	723	1 579	9 119	45 904
	Scaler	694	579	1 523	9 050	44 887
bloat	ci	32	1 924	2 014	8 939	61 150
	2obj	3 525	1 193	1 427	8 470	53 143
	Zipper^e-2obj	40	1 279	1 479	8 520	53 628
	introA-2obj	61	1 809	1 690	8 869	60 111
	introB-2obj	141	1 621	1 522	8 626	55 455
	Scaler	433	1 222	1 465	8 495	53 867

Program	Pointer analysis	Time (s)	#fail-cast	#poly-call	#reach-mtd	#call-edge
jython	ci	110	2 234	2 778	12 718	114 856
	2obj	>3h	—	—	—	—
	Zipper^e-2obj	134	1 781	2 486	12 026	107 113
	introA-2obj	417	2 202	2 632	12 663	114 095
	introB-2obj	>3h	—	—	—	—
	Scaler	495	1 852	2 500	12 167	107 410
hsqldb	ci	60	1 662	1 592	11 486	63 790
	2obj	>3h	—	—	—	—
	Zipper^e-2obj	119	1 032	1 261	10 440	51 261
	introA-2obj	110	1 558	1 482	11 367	60 333
	introB-2obj	313	1 034	1 260	10 378	51 278
	Scaler	388	1 120	1 197	10 639	52 063
pmd5	ci	71	2 948	4 183	15 254	104 457
	2obj	>3h	—	—	—	—
	Zipper^e-2obj	414	2 153	3 634	14 908	93 516
	introA-2obj	385	2 820	3 823	15 117	101 762
	introB-2obj	3 233	2 524	3 694	15 006	96 565
	Scaler	777	2 176	3 536	14 895	92 775
jedit	ci	112	3 382	4 749	21 006	118 426
	2obj	>3h	—	—	—	—
	Zipper^e-2obj	100	2 304	4 065	20 418	98 290
	introA-2obj	369	3 091	4 409	20 849	114 875
	introB-2obj	5 582	2 595	4 070	20 504	103 557
	Scaler	1 741	2 377	3 990	20 499	97 999
soot	ci	1 059	16 570	16 532	32 459	415 476
	2obj	>3h	—	—	—	—
	Zipper^e-2obj	>3h	—	—	—	—
	Zipper^e-2obj (0.5%)	797	10 673	14 666	31 965	326 092
	introA-2obj	1 428	16 503	15 947	32 390	413 083
	introB-2obj	6 365	15 474	14 895	32 222	319 431
	Scaler	1 358	10 549	14 822	31 982	374 877
eclipse-r	ci	141	4 190	9 197	20 862	161 222
	2obj	>3h	—	—	—	—
	Zipper^e-2obj	3 052	3 223	8 599	20 499	148 037
	introA-2obj	1 441	4 175	8 889	20 853	160 139
	introB-2obj	550	3 640	8 539	20 491	149 980
	Scaler	747	3 211	8 486	20 374	145 953
briss	ci	235	4 904	6 297	26 582	176 785
	2obj	>3h	—	—	—	—
	Zipper^e-2obj	217	3 158	5 306	25 537	151 550
	introA-2obj	628	4 889	6 076	26 507	175 565
	introB-2obj	>3h	—	—	—	—
	Scaler	1 387	3 428	5 323	25 652	152 761
h2	ci	51	1 866	3 946	14 192	95 541
	2obj	>3h	—	—	—	—
	Zipper^e-2obj	254	1 418	3 646	13 903	89 292
	introA-2obj	173	1 711	3 746	14 078	93 233
	introB-2obj	3 103	1 489	3 656	13 883	89 537
	Scaler	1 997	1 373	3 605	13 850	88 268
gruntspud	ci	185	3 583	5 703	24 887	148 874
	2obj	>3h	—	—	—	—
	Zipper^e-2obj	288	2 549	4 739	24 117	120 316
	introA-2obj	526	3 326	5 338	24 739	144 851
	introB-2obj	>3h	—	—	—	—
	Scaler	3 002	2 479	4 699	24 207	120 177
columba	ci	943	4 889	6 669	31 476	195 930
	2obj	>3h	—	—	—	—
	Zipper^e-2obj	837	3 444	5 213	26 051	140 637
	introA-2obj	3 356	4 582	6 249	31 100	187 401
	introB-2obj	>3h	—	—	—	—
	Scaler	329	2 897	4 799	25 729	123 330

PV serves as a criterion to select which methods may significantly degrade analysis efficiency. The smaller the PV, the more methods are considered efficiency-critical. Therefore, we can decrease PV to allow Zipper^e to identify and exclude more efficiency-critical methods so that fewer methods will be analyzed context-sensitively, thereby increasing the chance of scalability. In an extra experiment for `soot` (shown as Zipper^e-2obj (0.5%) in Table 8), we set PV to 0.5%, which allows Zipper^e-2obj to finish analyzing `soot` in only 797 seconds (even faster than CI for `soot`, which costs 1 059 seconds). This result demonstrates the flexibility of Zipper^e for balancing efficiency and precision.

Zipper^e-2obj vs. Scaler. The recent Scaler analysis [Li et al. 2018b] is an interesting comparison target for Zipper^e, since it follows a very different design but with similar goals. Scaler is a scalability-first analysis: it may not achieve full efficiency, nor full precision, but it will aim to avoid worst-case running-time blowup due to potentially enormous points-to sets. (More discussion can be found in Section 7.) Unlike introspective analyses and Zipper^e, which either apply one kind of context sensitivity (e.g., `2obj`) to a method or not, Scaler assigns different kinds of context sensitivity to different methods. As seen in Table 8, Scaler is generally much more expensive than Zipper^e, consistently with its design emphasis on avoiding *worst-case* behavior, rather than targeting the specific program elements responsible for keeping good precision while making the analysis as fast as possible. In two cases, however, (`eclipse-r` and `columba`) Scaler is both faster and (slightly) more precise than Zipper^e, suggesting potential for future improvement. Notably, the tuned Zipper^e-2obj (0.5%) on `soot` is both faster and more precise than Scaler, therefore more advanced tuning may be the avenue to such improvement. As summarized in Section 6.6.1, Zipper^e generally exhibits significantly better performance than Scaler with comparable precision.

Zipper^e-2obj Is Even Faster Than CI for Five Programs. According to existing literature, CI is the fastest pointer analysis for Java, and a context-sensitive pointer analysis, whether conventional or selective, is less efficient than CI [Bravenboer and Smaragdakis 2009; Hassanshahi et al. 2017; Jeon et al. 2018; Jeong et al. 2017; Kastrinis and Smaragdakis 2013; Lhoták and Hendren 2003; Li et al. 2018b; Milanova et al. 2005; Smaragdakis et al. 2011, 2014; Tan et al. 2016, 2017]. However, as shown in Table 8, we find that Zipper^e-2obj (as a selective context-sensitive pointer analysis) is faster than CI for five programs: `jpc`, `chart`, `jedit`, `briss` and `columba`. To the best of our knowledge, these results demonstrate for the first time that a context-sensitive pointer analysis can be faster than a context-insensitive one for a non-trivial portion of realistic programs. The issue is not one of trade-off between context-sensitive and context-insensitive analyses—after all, Zipper^e requires a CI pre-analysis in order to detect flow patterns. However, this surprising finding prompts some rethinking of the relationship between the extra cost of computing and maintaining context information and the saved cost in data propagation due to the improved precision of context sensitivity. The Zipper^e results show that CI is not an upper bound of speed for Java pointer analysis: for some programs it is possible to achieve better speed and precision simultaneously by choosing the right methods to be analyzed context-sensitively.

6.6.2 Zipper^e's Effectiveness as a Pre-Analysis. Table 9 shows the overhead of running Zipper^e. On average, Zipper^e spends 59 seconds on analyzing a program. This cost can be considered negligible compared with the significant speedup achieved by Zipper^e as shown in Table 8. Noticeably, Zipper^e spends more time on analyzing programs like `jython` and `columba`, which is not surprising as these programs are more complex or larger than the rest. For example, `jython` is a well-known hard-to-analyze program [Kastrinis and Smaragdakis 2013; Smaragdakis et al. 2014; Tan et al. 2017], and even if using context-insensitivity, `columba` analysis still costs close to 1 000 seconds (Table 8).

As Zipper^e further excludes methods that may hurt efficiency, fewer methods will be analyzed context-sensitively under its guidance. For all 20 evaluated programs, on average, Zipper reports

Table 9. The overhead of running Zipper^e.

Program	batik	checkstyle	sunflow	findbugs	jpc	eclipse	chart	fop	xalan	bloat
Zipper^e time (seconds)	26	8	20	6	14	3	7	18	5	4

Program	jython	hsqldb	pmd5	jedit	soot	eclipse-r	briss	h2	gruntspud	columba
Zipper^e time (seconds)	292	8	13	44	185	17	85	8	123	301

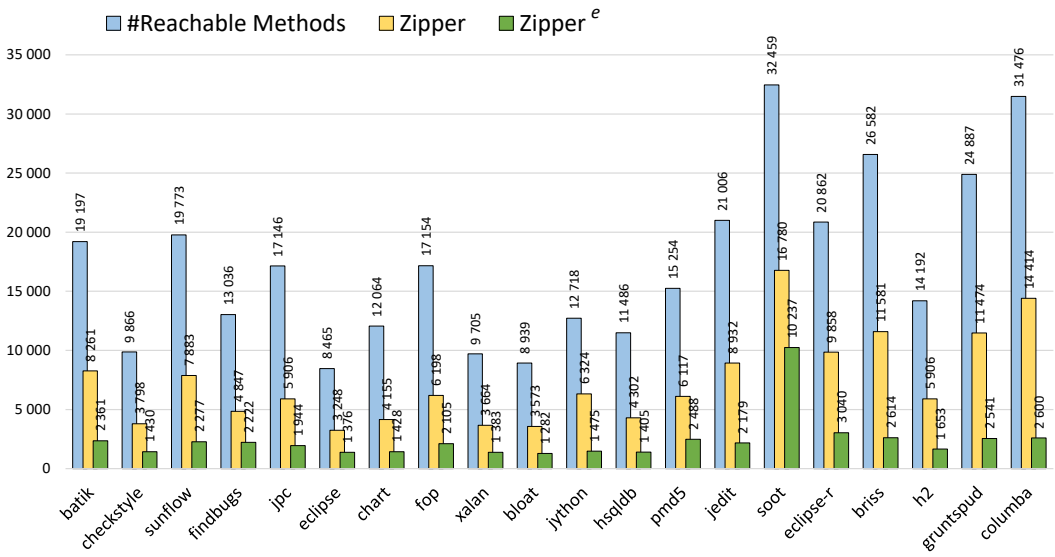

Fig. 13. The numbers of methods selected for context sensitivity by Zipper and Zipper^e.

41% of the methods as precision-critical, and by also considering efficiency-critical methods, Zipper^e reduces the number of methods to be analyzed context sensitively to 14%. The number of methods selected by Zipper and Zipper^e for each individual program is shown in Figure 13.

7 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 6.2 and Zipper^e in Section 6.6. Like Zipper and Zipper^e, 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 and Zipper^e, it relies on a set of six manually-selected metrics 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 Sections 6.2 and 6.6, 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; Zipper^e outperforms IntroA and IntroB in terms of both precision and efficiency in most 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], Zipper and Zipper^e, 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.

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]. Notably, 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 to be precision-critical. Instead, Zipper and Zipper^e are principled approaches derived from the insight of identifying the precision-loss patterns inherent in a program; thus their guiding is interpretable and their guided results are tractable, resulting in more stable performance.

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 and Zipper^e make a context-sensitive pointer analysis faster while preserving most of its precision.

Scaler [Li et al. 2018b] 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 perhaps be combined, using Scaler to estimate the context-sensitive points-to information only for the precision-critical methods identified by Zipper. As an approach to reducing analysis cost, unlike Scaler which may sacrifice noticeable precision for ensuring scalability, Zipper^e still considers precision (by extending Zipper) when selecting methods for context sensitivity. This explains why Zipper^e achieves very good precision for all evaluated programs as demonstrated in Section 6.6.

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. In contrast, Zipper and Zipper^e work 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. 2018b] 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 (and Zipper^e), we introduce precision flow graphs (PFGs) to express the three kinds of value flow patterns (Section 3) and identify the precision-critical methods by solving a graph reachability problem on the PFG (Section 4.3). 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 several key differences from PFGs. First, different from VFGs, the value flows in PFGs are defined on the basis of classes and their In and Out methods. Second, the direct flows in PFGs can already capture the value flows through load and store operations that are expressed as the indirect value flows in VFGs, but VFGs cannot express the wrapped/unwrapped flows in PFGs. Finally, pointer information is expressed differently in VFGs and PFGs.

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.

Demand-driven analyses (e.g., [Shang et al. 2012; Späth et al. 2016; Sridharan and Bodík 2006; Sridharan et al. 2005; Sui and Xue 2016; Wang et al. 2017; Yan et al. 2011]) typically only compute points-to information for program points that may affect a particular site of interest for specific clients. In contrast, the Zipper and Zipper^e analyses as well as other whole-program pointer analyses [Bravenboer and Smaragdakis 2009; Hassanshahi et al. 2017; Jeon et al. 2019, 2018; Jeong et al. 2017; Kastrinis and Smaragdakis 2013; Lhoták and Hendren 2003, 2006; Li et al. 2018b; Milanova et al. 2002, 2005; Smaragdakis et al. 2011, 2014; Tan et al. 2016, 2017; Thiessen and Lhoták 2017; Whaley and Lam 2004] compute points-to information for all sites, thereby providing information for all possible clients.

8 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. However, it is challenging to determine when a context-sensitive analysis will yield precision benefits, or, conversely, when omitting context sensitivity for a method would introduce imprecision.

By introducing a model based on three general patterns of value flow (direct, wrapped, and unwrapped flows), this article explains where and how most imprecision is introduced in a context-insensitive pointer analysis. The model provides a foundation to efficiently identify precision-critical methods in a principled way. Accordingly, we present two new selective context-sensitive pointer analyses for Java, Zipper and Zipper^e. Both analyses are conceptually simple and easy to integrate with existing pointer-analysis tools.

Extensive evaluation on standard benchmarks and real-world Java programs demonstrates the effectiveness of the analyses. On average, Zipper preserves essentially all of the precision (98.8%) of a highly-precise conventional context-sensitive pointer analysis (`2obj`), with a substantial speedup (on average, 3.4× and up to 9.4×). As a result, one can consider using Zipper as a full replacement of conventional highly-precise context-sensitive pointer analysis. In addition, Zipper and its implementation provide a basis to further investigate the relation between context sensitivity and the precision benefits it brings. Zipper^e is a prime outcome of such investigation: it preserves 94.7% of the precision of `2obj`, with an order-of-magnitude speedup (on average, 25.5× and up to 88×). In addition, on 10 programs for which `2obj` fails to terminate (within 3 hours), Zipper^e can guide `2obj` to analyze them in less than 11 minutes on average, and with very good precision. The experimental evaluation of Zipper^e further shows that, for several programs, it is even possible to simultaneously achieve faster and more precise pointer analysis than a context-insensitive approach.

We believe that these results establish Zipper and Zipper^e as new sweet spots in the well-established pointer analysis trade-off of precision and efficiency. Furthermore, we expect that the approach offers interesting insights, leading into deep understanding of how context-sensitive analysis achieves its significant precision gains.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful comments. This work was supported in part by National Key R&D Program (Grant #2017YFB1001801) and National Natural Science Foundation (Grant #61690204) of China, and by the European Research Council (ERC) under the FP7 and Horizon 2020 research and innovation programs (grant agreements 307334, 790340, and 647544). The authors would also like to thank the support from the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

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

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

Minseok Jeon, Sehun Jeong, Sungdeok Cha, and Hakjoo Oh. 2019. A Machine-Learning Algorithm with Disjunctive Model for Data-Driven Program Analysis. *ACM Trans. Program. Lang. Syst.* 41, 2 (2019), 13:1–13:41. https://doi.org/10.1145/3293607

Minseok Jeon, Sehun Jeong, and Hakjoo Oh. 2018. Precise and Scalable Points-to Analysis via Data-driven Context Tunneling. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 140 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276510

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. 2018a. Precision-guided Context Sensitivity for Pointer Analysis. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 141 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276511

Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018b. 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)*. ACM, 129–140. https://doi.org/10.1145/3236024.3236041

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

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

Lei Shang, Xinwei Xie, and Jingling Xue. 2012. On-demand dynamic summary-based points-to analysis. In *10th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2012, San Jose, CA, USA, March 31 - April 04, 2012*. ACM, 264–274. https://doi.org/10.1145/2259016.2259050

Micha Sharir and Amir Pnueli. 1981. *Two approaches to interprocedural data flow analysis*. Prentice-Hall, Chapter 7, 189–234.

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 & 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

Yulei Sui and Jingling Xue. 2016. On-demand Strong Update Analysis via Value-flow Refinement. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 460–473. https://doi.org/10.1145/2950290.2950296

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

Manas Thakur and V. Krishna Nandivada. 2019. Compare Less, Defer More: Scaling Value-contexts Based Whole-program Heap Analyses. In *Proceedings of the 28th International Conference on Compiler Construction (CC 2019)*. ACM, New York, NY, USA, 135–146. https://doi.org/10.1145/3302516.3307359

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

John Whaley and Monica S. Lam. 2004. Cloning-based Context-sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04)*. ACM, New York, NY, USA, 131–144. https://doi.org/10.1145/996841.996859

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

Dacong Yan, Guoqing (Harry) Xu, and Atanas Rountev. 2011. Demand-driven context-sensitive alias analysis for Java. In *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*. ACM, 155–165. https://doi.org/10.1145/2001420.2001440