SHARP: Fast Incremental Context-Sensitive Pointer Analysis for Java

BOZHEN LIU, Texas A&M University, USA
JEFF HUANG, Texas A&M University, USA

We present SHARP, an incremental context-sensitive pointer analysis algorithm that scales to real-world large complex Java programs and can also be efficiently parallelized. To our knowledge, SHARP is the first algorithm to tackle *context-sensitivity* in the state-of-the-art incremental pointer analysis (with regards to code modifications including both statement *additions* and *deletions*), which applies to both *k-CFA* and *k-obj*. To achieve it, SHARP tackles several technical challenges: soundness, redundant computations, and parallelism to improve scalability without losing precision. We conduct an extensive empirical evaluation of SHARP on large and popular Java projects and their code commits, showing impressive performance improvement: our incremental algorithm only requires on average 31 seconds to handle a real-world code commit for *k-CFA* and *k-obj*, which has comparable performance to the state-of-the-art incremental context-insensitive pointer analysis. Our parallelization further improves the performance and enables SHARP to finish within 18 seconds per code commit on average on an eight-core machine.

CCS Concepts: • **Theory of computation → Design and analysis of algorithms**; • **Computing methodologies → Parallel algorithms**.

Additional Key Words and Phrases: incremental pointer analysis, context-sensitive algorithm, parallelization

ACM Reference Format:
Bozhen Liu and Jeff Huang. 2022. SHARP: Fast Incremental Context-Sensitive Pointer Analysis for Java. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 88 (April 2022), 28 pages. https://doi.org/10.1145/3527332

1 INTRODUCTION

Pointer analysis, serving as the core of most compiler optimizations, is a fundamental problem in static program analysis. Researchers have proposed a wide spectrum of classical algorithms [Hardekopf and Lin 2011; Lhoták 2002; Lhoták and Hendren 2008; Méndez-Lojo et al. 2010, 2012; Milanova et al. 2005; Shivers 1991; Sridharan and Bodík 2006; Sridharan et al. 2005; Whaley and Lam 2004], however, it remains challenging to scale pointer analysis to large codebases with good precision.

Recently, Liu et al. [Liu and Huang 2018; Liu et al. 2019] propose a promising algorithm that computes the points-to results *incrementally* without losing precision. The key is to observe a change-locality property of Andersen's analysis [Andersen 1994]: the updates of points-to results upon a code change can be determined by the local neighbors of the change in the pointer assignment graph (PAG) [Lhoták 2002]. This allows developing a sound incremental pointer analysis that only recomputes the change impact by leveraging the memoized intermediate analysis results, instead of re-running an exhaustive pointer analysis for every code change. Their algorithm has scaled to large complex Java programs with averaged response time under a second (per statement change),

Authors' addresses: Bozhen Liu, Texas A&M University, USA, april1989@tamu.edu; Jeff Huang, Texas A&M University, USA, jeffhuang@tamu.edu.

2475-1421/2022/4-ART88
https://doi.org/10.1145/3527332

```
T1:
1  public void foo() {
2    A a1 = new A();//O1
3    B b1 = new B();//O2
4    a1.m1(b1);//S1
5  }
T2:
6  public void bar() {
7    A a2 = new A();//O3
8    B b2 = new B();//O4
9    a2.m1(b2);//S2
10 }
```

```
11 public class A {
12   public B f;

13   public void m1(B p1){
14     f = p1;
15     if(...){
16 -     f = new B();//O5       ①
17     }
18 -   m2(f);//S3              ②
19 +   m3(p1);//S6             ③
20     ...
21   }
```

```
22   public void m2(B p2) {
23     m3(p2);//S4  ...
24   }

25   public void m3(B p3) {
26     m4(p3);//S5  ...
27   }

28   public void m4(B p4) {
29 +   x = p4; ...             ④
30   }
31 }
```

Fig. 1. A Java example with four statement changes: ①-④.

making it suitable to be applied in the programming phase to detect sophisticated bugs such as data races.

However, their incremental algorithm only applies to context-insensitive pointer analysis, and hence can be imprecise for many applications. More importantly, it is unclear how to extend their algorithm with context-sensitivity. To illustrate the problem, consider the example in Figure 1 involving two threads. Before the four highlighted changes (delete statement ①② and add statement ③④), if we run an incremental static data race detector, D4 [Liu and Huang 2018], it reports five warnings on field *f*: lines (14,14); lines (14,15); lines (14,16); lines (15,15); lines (15,16). Unfortunately, these are all false positives because D4 is based on context-insensitive pointer analysis, which fails to distinguish the two instances of *f* from different call sites: o_2 from $S1$ and o_4 from $S2$. The strength of D4 is that when the developer pushes code commits it can instantly detect new races or invalidate old warnings. After the four code changes ①-④, D4 still reports one false positive on lines (14,14).

It would be highly desirable for developers to have a fast debugging tool that works incrementally like D4 but with much higher precision, since false positives can significantly reduce the debugging efficiency. To achieve a better precision, *k-CFA* [Shivers 1991] adopts a *k-call-site-sensitive* algorithm with a context-sensitive heap during the on-the-fly call-graph (CG) construction [Lhoták 2002]. However, *k-CFA* is known to be unscalable [Smaragdakis et al. 2014a; Tan et al. 2017]; this applies to other context-sensitive algorithms such as *k-object-sensitive* [Milanova et al. 2005] (denoted *k-obj*) and *k-type-sensitive* [Smaragdakis et al. 2014b] (denoted *k*-type).

Our goal is to tackle the scalability challenge for context-sensitivity with an incremental algorithm. We are facing the following technical challenges:

- How to incrementally compute points-to results with context-sensitivity?
- If one could come up with an incremental context-sensitive pointer analysis, how to guarantee its soundness? The added and *especially deleted* pointers, objects, and method calls must all be handled correctly together with their contexts.
- How to generalize the incremental algorithm for different types of contexts, *i.e.*, *k-CFA* and *k-obj*?

In this paper, we present SHARP, a novel incremental algorithm that addresses these challenges. Towards pushing the performance boundary of context-sensitive pointer analysis, this work makes several important contributions:

(1) The first contribution is a new algorithm that enhances the existing incremental algorithms [Liu and Huang 2018; Liu et al. 2019] with context-sensitivity of *k-CFA* and *k-obj*. In particular, we discover the redundant computation in the fixed-point-based pointer analysis when a deleted new statement or method invocation destroys a sequence of method calls and

Table 1. On-the-fly Andersen's Algorithm (context-insensitive) constructs PAG = (N_{pag}, E_{pag}) and CG = (N_{cg}, E_{cg}) iteratively until reaching a fixed point.

Type		Statement	PAG Edge	Constraint
New		$x = new\ C()$	$o \rightarrow x$	$o_c \in pts(x)$
Assign		$x = y$	$y \rightarrow x$	$pts(y) \subseteq pts(x)$
Load	Field	$x = y.f$	$y \overset{load[f]}{\rightsquigarrow} x$; $\forall o \in pts(y)$: $o.f \rightarrow x$	$\forall o \in pts(y)$: $pts(o.f) \subseteq pts(x)$
	Array	$x = y[i]$	$y \overset{load*}{\rightsquigarrow} x$; $\forall o \in pts(y)$: $o.* \rightarrow x$	$\forall o \in pts(y)$: $pts(o.*) \subseteq pts(x)$
Store	Field	$x.f = y$	$y \overset{store[f]}{\rightsquigarrow} x$; $\forall o \in pts(x)$: $y \rightarrow o.f$	$\forall o \in pts(x)$: $pts(y) \subseteq pts(o.f)$
	Array	$x[i] = y$	$y \overset{store*}{\rightsquigarrow} x$; $\forall o \in pts(x)$: $y \rightarrow o.*$	$\forall o \in pts(x)$: $pts(y) \subseteq pts(o.*)$
Invoke //called from $m'()$		$x = y.g(..., a_i, ...)$	$y \overset{invoke\ g()}{\rightsquigarrow}$; $\forall o \in pts(y)$: $m = dispatch(o, g)$ $E_{cg} = E_{cg} \cup \{m' \rightarrow m\}$, $y \rightarrow this_m$, $a_i \rightarrow p_i$, $r_m \rightarrow x$	$pts(y) \subseteq pts(this_m)$ $pts(a_i) \subseteq pts(p_i)$ $pts(r_m) \subseteq pts(x)$

contexts. We utilize two common properties in both contexts: *inheritance* and *uniqueness*, and leverage them to identify and remove all invalid program elements in advance to avoid useless propagation of changes generated by iteratively identified invalid method calls.

(2) The second contribution is a parallel algorithm that further scales the proposed incremental algorithm by leveraging a third property, *convergence*. Besides, we study the incremental impact on the PAG from real-world code commits, and discuss various parallel scenarios from the perspective of efficiency, redundancy and conflict. Instead of pre-ordering multiple changed statements [Saha and Ramakrishnan 2006; Su et al. 2014], our parallel algorithm minimizes conflict while avoiding redundant computation.

(3) The third contribution is an extensive evaluation of SHARP on real-world code commits. We implemented an end-to-end incremental *k-CFA* and *k-obj* algorithm, and conducted an empirical evaluation on a collection of popular and actively maintained Java projects on GitHub, *i.e.*, Hbase, Lucene, Yarn, Zookeeper. The evaluation shows similar efficiency and scalability as the state-of-the-art context-insensitive incremental algorithm [Liu and Huang 2018; Liu et al. 2019]: SHARP requires 31 s on average to handle all the statement changes from a real-world git commit; our parallel algorithm achieves on average 1.3x speedup over our incremental algorithm on an eight-core machine.

The rest of the paper is organized as follows. Section 2 illustrates the existing incremental algorithm [Liu and Huang 2018; Liu et al. 2019] and discusses its scope and limitations. Section 3 presents our incremental algorithm. Section 4 presents our parallel algorithm, which can be applied in both context-sensitive and -insensitive pointer analysis. Section 5 presents our empirical evaluation. Section 6 discusses related work and Section 7 concludes this paper.

$$\textsc{DeletePAGEdge}\ \frac{e_{pag} \in E_{pag}}{E_{pag} \setminus \{e_{pag}\}}$$

$$\begin{cases} \boldsymbol{if}\, e_{pag} = o \longrightarrow x \vee y \longrightarrow x : \textsc{DeletePointsTo}(e_{pag}, pts(y)) \\ \boldsymbol{if}\, e_{pag} = y \overset{load[f]}{\rightsquigarrow} x \vee y \overset{load*}{\rightsquigarrow} x : \textsc{DeleteLoad}(e_{pag}, pts(y)) \\ \boldsymbol{if}\, e_{pag} = y \overset{store[f]}{\rightsquigarrow} x \vee y \overset{store*}{\rightsquigarrow} x : \textsc{DeleteStore}(e_{pag}, pts(x)) \\ \boldsymbol{if}\, e_{pag} = y \overset{invoke\ g()}{\rightsquigarrow}: \textsc{DeleteInvoke}(e_{pag}, pts(y)) \end{cases}$$

$$\textsc{DeletePointsTo}\ \frac{e_{pag} = o \longrightarrow x \vee y \longrightarrow x \qquad \Delta}{E_{pag} \setminus \{e_{pag}\} \qquad \textsc{Check}(x, \Delta)}$$

$$\textsc{DeleteLoad}\ \frac{y \overset{load[f]}{\rightsquigarrow} x \vee y \overset{load*}{\rightsquigarrow} x \qquad \Delta}{\forall o \in \Delta : \begin{cases} \textsc{Field}: \ \textsc{DeletePointsTo}(o.f \longrightarrow x, pts(o.f)) \\ \textsc{Array}: \textsc{DeletePointsTo}(o.* \longrightarrow x, pts(o.f)) \end{cases}}$$

$$\textsc{DeleteStore}\ \frac{y \overset{store[f]}{\rightsquigarrow} x \vee y \overset{store*}{\rightsquigarrow} x \qquad \Delta}{\forall o \in \Delta : \begin{cases} \textsc{Field}: \ \textsc{DeletePointsTo}(y \longrightarrow o.f, pts(y)) \\ \textsc{Array}: \textsc{DeletePointsTo}(y \longrightarrow o.*, pts(y)) \end{cases}}$$

$$\textsc{DeleteInvoke}\ \frac{y \overset{invoke\ g()}{\rightsquigarrow} \qquad \Delta}{\boldsymbol{if}\ deleted, E_{cg} \setminus \{m' \relbar\!\!\blacktriangleright m\}}$$

$$\forall o \in \Delta : \begin{cases} m = dispatch(o, g), \\ \textsc{DeletePointsTo}(y \longrightarrow this_m, pts(y)), \\ \textsc{DeletePointsTo}(a_i \longrightarrow p_i, pts(a_i)), \\ \textsc{DeletePointsTo}(r_m \longrightarrow x, pts(r_m)) \end{cases}$$

$$\textsc{Check}\ \frac{\Delta \neq \emptyset \qquad y \text{ is a node that } \Delta \text{ propagates to}}{\forall u \longrightarrow y \in E_{pag} : \Delta = \Delta \setminus (\Delta \cap pts(u))}$$

$$\textsc{Propagate}\ \frac{pts(y) = pts(y) \setminus \Delta \qquad\qquad \Delta \neq \emptyset}{\forall y \longrightarrow v \in E_{pag} : \textsc{Check}(v, \Delta)}$$

$$\forall e_{pag} = y \overset{load[f]}{\rightsquigarrow} w \in E_{pag} : \textsc{DeleteLoad}(e_{pag}, \Delta)$$
$$\forall e_{pag} = w \overset{store[f]}{\rightsquigarrow} y \in E_{pag} : \textsc{DeleteStore}(e_{pag}, \Delta)$$
$$\forall e_{pag} = y \overset{invoke\ g()}{\rightsquigarrow} \in E_{pag} : \textsc{DeleteInvoke}(e_{pag}, \Delta)$$

Fig. 2. IPA: DeletePAGEdge: The inference rules of handling a deleted points-to or dynamic edge in the PAG = (N_{pag}, E_{pag}) and CG = (N_{cg}, E_{cg}).

2 EXISTING ALGORITHMS

We first review the key insight of the existing incremental algorithm for context-insensitive pointer analysis [Liu and Huang 2018; Liu et al. 2019] (denoted *IPA*) as well as its parallel algorithm (denoted *PIPA*).

2.1 Background

Table 1 describes the rules for context-insensitive on-the-fly Andersen's algorithm for Java programs, of which points-to relationships are represented by a pointer assignment graph (PAG) [Lhoták 2002] and calling relationships between methods are represented by a call graph (CG).

There are five types of statements that determine the points-to relationships in the table. New and Assign are simple statements which only introduce *points-to edges* (denoted $\longrightarrow$). An abstract heap object is represented by o, and a points-to edge $y \longrightarrow x$ indicates the inclusion-based constraint between the points-to sets of y and x (denoted $pts(y) \subseteq pts(x)$). Load, Store and Invoke are complex statements due to the *dynamic edges* (denoted $\rightsquigarrow$), of which points-to edges vary with the points-to set changes of its base variable (*i.e.*, y for Load and Invoke, x for Store).

The rules for array load/store are similar to the ones for field load/store: we abstract each array with a single field $*$ and considering all array accesses as to that single field. As most commonly adopted pointer analysis algorithms, we do not distinguish different array indexes.

Invoke is the type of statement that directly builds the CG by virtual dispatch: given an abstract heap object o that the base variable y point to, a dispatch function is invoked to resolve the virtual dispatch of $g()$ on o to a target method m. This creates new CG edges (denoted $m' \twoheadrightarrow m$) and new points-to constraints for parameters ($y \longrightarrow this_m$ and $a_i \longrightarrow p_i$), return value ($r_m \longrightarrow x$) and statements enclosed in the target method.

2.2 Incremental Algorithm

The key contribution of IPA is to minimize the redundant computation in the existing algorithms (*reset-recompute* [Saha and Ramakrishnan 2005b, 2006], and *reachability-based* [Shang et al. 2012; Zhan and Huang 2016]) when solving for incremental statement deletions. We refer the readers to the original papers [Liu and Huang 2018; Liu et al. 2019] for the difficulties in handling deletion. In this section, we re-formulate the algorithms of IPA by using inference rules to make the later reference and comparison more convenient and consistent.

IPA takes statement changes as input and efficiently updates the change-affected part in the PAG and CG. For each deleted statement, IPA extracts the PAG edge(s) according to Table 1 and follow the inference rules shown in Figure 2 to handle each deleted PAG edge. The procedure of resolving incremental statement additions is similar but simpler, so this section focuses on the illustration of how to handle incremental deletions.

DeletePAGEdge takes a deleted PAG edge e_{pag} as input, remove this edge from the PAG, and calls different rules according to the type of e_{pag}. The called rules (*i.e.*, DeletePointsTo, DeleteLoad, DeleteStore and DeleteInvoke) take e_{pag} and a set of points-to set changes Δ as input, where Δ is initialized by this rule.

DeletePointsTo is applied to a points-to edge e_{pag}, which removes the edge from PAG and calls Check. The key insight of IPA is **Check** and **Propagate**: the former rule confirms the real change Δ in $pts(y)$ by checking the points-to sets along all the incoming edges of y; while the later rule abstracts how to propagate Δ along the outgoing edges of y. Both of the rules guarantee the soundness and efficiency of IPA after any statement deletion.

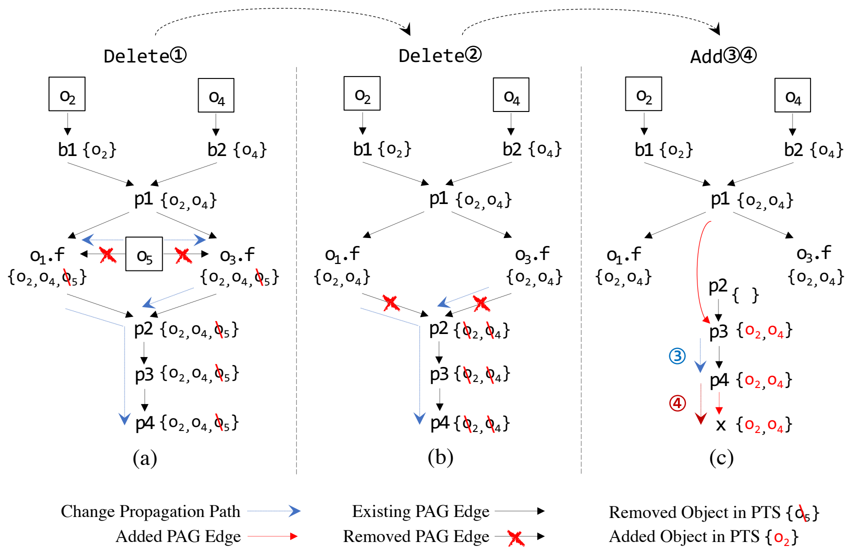


Fig. 3. The context-insensitive PAG changes for code changes ①②③④ in Figure 1.

For the three complex statements that introduce dynamic edges to the PAG, IPA has DeleteLoad, DeleteStore and DeleteInvoke rules. Note that the dynamic edge is kept when a change is propagated but no statement is deleted.

DeleteLoad is called on a deleted dynamic edge generated by a Load Field or Load Array statement: the dynamic edge (*i.e.*, $y \overset{load[f]}{\rightsquigarrow} x$ for Field or $y \overset{load*}{\rightsquigarrow} x$ for Array) and its derived points-to edges (*i.e.*, $o.f \longrightarrow x$ for Field or $o.* \longrightarrow x$ for Array) are deleted, where $o \in \Delta$ is removed from $pts(y)$. If this rule is called by Propagate, the dynamic edge $y \rightsquigarrow x$ is kept in the PAG. However, $o \in \Delta$ is no longer valid in $pts(y)$ so that the derived points-to edges should be deleted.

Similarly, **DeleteStore** is applied on a deleted dynamic edge from Store Field or Store Array statements, or propagating a change Δ for such a dynamic edge from y to x.

DeleteInvoke updates both PAG and CG. For a deleted dynamic edge generated by a Invoke statement, this rule requires the deletion of both the invalid calling edges $m' \twoheadrightarrow m$ from the CG, as well as the introduced points-to edges in the PAG. After the change propagation from the deleted PAG edges, all other program elements generated by the callee method $m()$ in the PAG and CG remain the same.

IPA requires acyclic PAGs, which can be satisfied by the strongly connected component (SCC) optimization [Hardekopf and Lin 2007]. Because SCCs can be collapsed or broken by code changes, IPA also proposes an incremental algorithm to maintain SCCs dynamically on the PAG based on Tarjan's two-way search algorithm [Bender et al. 2015].

2.3 Illustration of IPA

We use the statement changes ①-④ in the code shown by Figure 1 to illustrate the rules of IPA. As depicted in Figure 3, IPA handles statement deletions ①② at first, and then additions ③④.

- **Delete ① `f = new B()`:** IPA extracts the PAG edges, calls DeletePAGEdge and removes edges $o_5 \longrightarrow o_1.f$ and $o_5 \longrightarrow o_3.f$. Then DeletePointsTo is called with the initialized change $\Delta = \{o_5\}$. Check and Propagate are performed on the removed edges, sequentially.

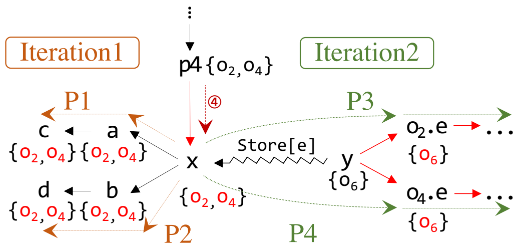


Fig. 4. The parallel propagation after adding statement `x = p4`.

Firstly, IPA applies Check rule to $o_1.f$ by checking whether any points-to set of PAG nodes (*i.e.*, $p1$) that point to $o_1.f$ contains o_5. After finding out $pts(p1)$ does not contain o_5, IPA confirms the change of $pts(o_1.f)$ is $\Delta = \{o_5\}$, and removes Δ from $pts(o_1.f)$. Next is to apply Propagate rule by propagating Δ to $p2$. Then the Check and Propagate are applied to $p2$, $p3$ and $p4$, successively.

Finally, the same procedure is applied to $o_3.f$. IPA removes $\Delta = \{o_5\}$ from $pts(o_3.f)$ and propagates to $p2$. However, o_5 is already removed from $pts(p2)$, so this propagation stops here.

- **Delete ② `m2(f)`:** According to DeleteInvoke, IPA deletes the edges $o_1.f \longrightarrow p2$ and $o_3.f \longrightarrow p2$ which represent the points-to relation between the actual and formal parameters for the removed method call to $m2()$. Both deleted edges have the change initialized to $\Delta = \{o_2, o_4\}$. Then IPA runs Check on $p2$ and finds out it has no incoming PAG edges. Hence, the change $\Delta = \{o_2, o_4\}$ is confirmed and removed from $pts(p2)$, and then propagated to $p3$. Afterwards, the Check and Propagate are applied on $p3$ and $p4$ in order.
- **Add ③ `m3(p1)`:** An edge $p1 \longrightarrow p3$ is added in Figure 3(c) for the parameter of method $m3()$. This is based on the Andersen's algorithm for Invoke statements in Table 1. Simpler than handling a deletion, IPA checks whether $pts(p3)$ contains the change $\Delta = pts(p1) = \{o_2, o_4\}$, adds Δ to $pts(p3)$ and propagates Δ to $p4$.
- **Add ④ `x = p4`:** IPA adds the PAG edge $p4 \longrightarrow x$ by following the rule for Assign statement and propagates the change $\Delta = pts(p4) = \{o_2, o_4\}$ to x .

2.4 Parallel Algorithm

PIPA is based on a *change consistency property* to run Propagate in parallel within each iteration of the fixed-point-based pointer analysis. Consider the PAG shown in Figure 4. After adding statement ④, PIPA confirms the change $\Delta = \{o_2, o_4\}$ in $pts(x)$ through Check and starts to propagate Δ to all the outgoing neighbours of x in parallel.

In the first iteration, PIPA applies Propagate on all the outgoing points-to edges of x in parallel, *i.e.*, the propagation along paths *P1* and *P2* are parallelized. Afterwards, each iteration propagates along one dynamic edge for complex statements. The second iteration handles $y \xrightarrow{store[e]} x$ introduced by statement `x.e = y`. PIPA creates two new points-to edges, $y \longrightarrow o_2.e$ and $y \longrightarrow o_4.e$ due to the points-to set change in x. Then the change $\Delta = pts(y) = \{o_6\}$ is propagated along *P3* and *P4* in parallel.

However, the parallel algorithm leaves plenty of room for improvement. For example, can the changes be propagated along all the paths, *e.g.*, *P1* - *P4*, in parallel? Can multiple statement changes be handled in parallel? If possible, can the multiple statements come from one method or different

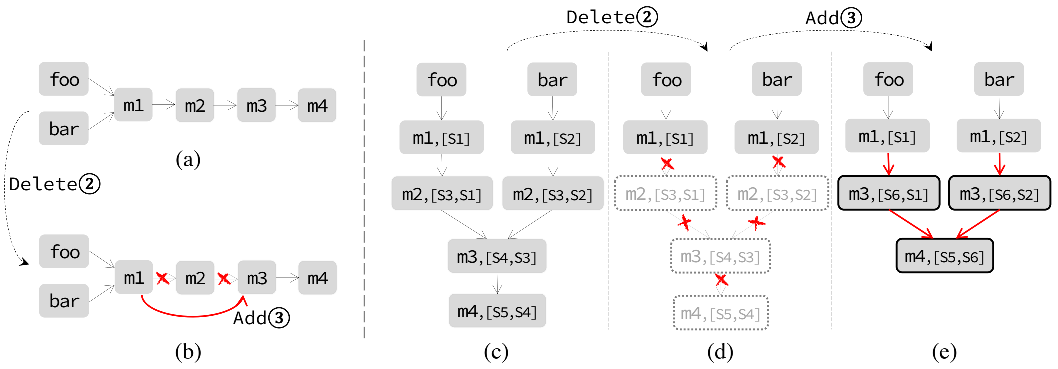


Fig. 5. The CG after each statement change in Figure 1. (a)(b) Context-insensitive CG. (c)-(e) 2-CFA CG.

methods? Is there any conflict or redundancy while updating points-to sets during the propagation? Moreover, is it possible to add more parallelization for incremental context-sensitive algorithms?

2.5 Scope and Limitations of IPA

Correct context is crucial while building PAG and CG for *k-CFA* and *k-obj*, which cannot be randomly decided for a statement change. Except for context, there exist redundant computations when applying IPA to context-sensitive algorithms.

2.5.1 More Computation for Method Call Changes. For a deleted/added method call statement in method *m*, we need to handle the change for multiple CG nodes of *m* but with different contexts. Moreover, this introduces more invalid/newly added CG nodes when contexts are considered, which involves more points-to constraints. Hence, handling a method call change requires more computation in context-sensitive algorithms than the one in IPA.

For example, replacing ② by ③ is simple for the context-insensitive CG as shown in Figure 5(a)(b) according to IPA: for the deletion of ②, the call edges $m1 \to m2 \to m3$ are removed. Now $m3$ has no caller, so nodes $m3$, $m4$ and their connected edge become invalid which should be removed. Nevertheless, IPA keeps them in case that a future change (adding ③) will add them back. So do the pointers and edges created by $m3$ and $m4$ in the PAG. This *reuse trick* is only valid in context-insensitive algorithm, because multiple calls share the same target method. In summary, only one node is removed from CG for a method call replacement.

However, for 2-CFA, deleting ② causes four nodes removed from its CG as shown in Figure 5(d), and adding ③ creates three new nodes in Figure 5(e). Here, the reuse trick cannot be adopted, because the deleted contexts can never be retrieved by later changes, unless the previous deletion of ② is withdrawn. In total, seven nodes are changed, not to mention the consequent update in the 2-CFA PAG.

Context-sensitive algorithms require extra computation to create/delete nodes, constraints with the incremental rules in order to guarantee the correctness. Therefore, we need an efficient algorithm to update PAG and CG together for a method call statement.

2.5.2 Redundant Computation for Deletion. For *k-CFA*, a deleted method call statement may introduce multiple invalid call sites that will be gradually discovered in the iterative computation between PAG and CG. If applying Check and Propagate directly, it conducts redundant change propagation to the pointers that later will be identified as invalid, because their contexts contain newly discovered, invalid call sites.

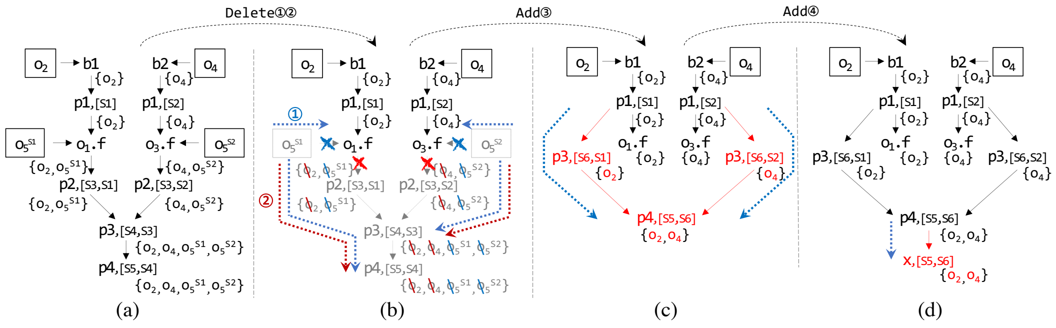
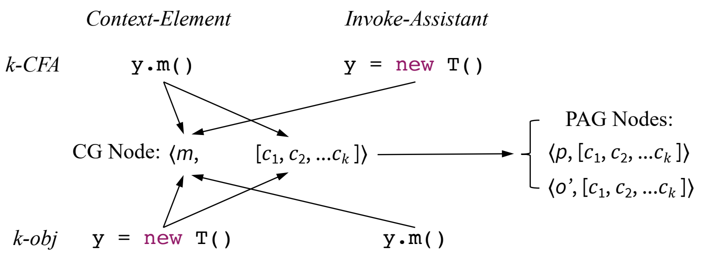


Fig. 6. The 2-CFA PAG after each statement change in Figure 1.

```
public void m'() {
    y = new T();//O
    y.m();//S
    …
}

public void m() {
    T p = new T();//O'
    …
}
```

(a)



(b)

Fig. 7. How New and Invoke determine the CG and PAG for *k-CFA* and *k-obj*, respectively. (a) An example code. (b) The Context-Element and Invoke-Assistant Statements for *k-CFA* and *k-obj*, and how they affect the CG and PAG.

Consider the deletion ② in Figure 5(c)(d), which makes $S3$ an invalid call site. As a result, methods $\langle m2, [S3, S1]\rangle$ and $\langle m2, [S3, S2]\rangle$ are invalid and removed. Then their parameter constraints are solved as shown in Figure 6(b): IPA resets $pts(\langle p2, [S3, S1]\rangle)$ and $pts(\langle p2, [S3, S2]\rangle)$ and propagates the changes $\{o_2, {o_5}^{S1}\}$ and $\{o_4, {o_5}^{S2}\}$, respectively. $pts(\langle p3, [S4, S3]\rangle)$ and $pts(\langle p4, [S5, S4]\rangle)$ are updated twice, each for one change as shown in Figure 6(b) (the first time is updated by the blue path, and the second time is updated by the maroon path). Later, IPA deletes the statements enclosed in the two removed methods, which removes CG edges $\langle m2, [S3, S1]\rangle \rightarrow \langle m3, [S4, S3]\rangle$ and $\langle m2, [S3, S2]\rangle \rightarrow \langle m3, [S4, S3]\rangle$. Now, $S4$ is discovered to be invalid, which indicates the previous change propagations are redundant. A more efficient way is to simply empty their points-to sets instead of running the Check and Propagate rules.

The same scenario exists in *k-obj* when deleting a new statement. To avoid such redundancy, we must discover all the invalid graph elements (*i.e.*, methods, contexts, pointers and objects) right after the deletion of a method call or new statement and before the propagation of points-to set changes.

3 SHARP

We now introduce, Sharp, our new incremental algorithm designed for *k-CFA* and *k-obj*. Here, we focus on how to efficiently handle statement deletions, which is challenging for incremental analyses [Liu et al. 2019; Saha and Ramakrishnan 2006; Zhan and Huang 2016]. Handling additions is then straightforward.

3.1 Definitions

A precise pointer analysis requires iterative computation between PAG and CG, in order to achieve a precise result. This introduces more computation workload when considering contexts. For Java programs, there are three facts that determine a context-sensitive CG node: (1) an Invoke statement, (2) the abstract heap object that the base variable of the invocation may point to and (3) context.

A context C is composed of a sequence of context elements c, *i.e.*, $C = [c_1, c_2, ...c_i, ...]$. In practice, to improve the scalability of context-sensitive pointer analysis, the length of context elements is bounded by a small constant k (*i.e.*, *k-limiting*). The most recent k context elements form the context at a method call.

A specific type of statement defines the context elements for a particular context-sensitive algorithm as shown in Figure 7: an Invoke statement defines a call site S as the context element of *k-CFA*, and a New statement defines an abstract heap allocation site O as the one of *k-obj*. We define this specific type of statement the *Context-Element Statement* (denoted Sctx).

Except for the Sctx, there is another type of statements determines the callee methods. *k-CFA* requires Invoke to define its contexts and the assistance of New to determine the dispatch of method invocation statements. For *k-obj*, it requires New to define its contexts and the assistance of Invoke to determine the dispatched target. We define this assistance statements the *Invoke-Assistant Statement* (denoted Sinv).

In this paper, we overload the $\in$ operator to more than set operations, for example: $s \in \{\text{New}\}$ (statement s is a New statement), Sctx $\in m$ (statement Sctx is from its enclosing method m), $c \in C$ (context element c is an element of context C).

Definition 3.1 (Context-Element Statement)

$$\textit{Sctx} = (s \in \{\textit{Invoke}\} \wedge \textit{k-CFA}) \vee (s \in \{\textit{New}\} \wedge \textit{k-obj})$$

Definition 3.2 (Invoke-Assistant Statement)

$$\textit{Sinv} = (s \in \{\textit{New}\} \wedge \textit{k-CFA}) \vee (s \in \{\textit{Invoke}\} \wedge \textit{k-obj})$$

To be specific, as shown in Figure 7(b), Sctx refers to Invoke statements and Sinv refers to New statements for *k-CFA*. While it is the opposite for *k-obj*: Sctx refers to New and Sinv refers to Invoke. No matter which context is selected in pointer analysis, both New and Invoke statements determine a CG node, which further defines a set of PAG nodes. Hence, we need to carefully handle the deletion of New and Invoke statements if we want to efficiently identify invalid graph elements.

3.2 Properties

We leverage two properties in context-sensitive CG:

- Inheritance: a caller node $\langle m', [c_1, c_2, ...c_k]\rangle$ must share the first $(k-1)$ context elements (*i.e.*, call site s in *k-CFA* and object allocation site o in *k-obj*) with its callee node $\langle m, [c, c_1, ...c_{k-1}]\rangle$.
- Uniqueness: one Sctx describes its unique context element c, and defines a set of unique contexts C where $c \in C$. C then defines a unique CG node $\langle m, C\rangle$ together with method m, and a unique set of pointer nodes $\langle y, C\rangle$, object nodes $\langle o, C\rangle$ and edges in PAG.

According to Inheritance, if c becomes invalid, all the nodes in CG with $c \in C$ become invalid, too. Due to Uniqueness, we can retrieve a set of nodes in CG and PAG that are derived from c. These two properties guide Sharp to immediately identify the invalid node(s) in CG and PAG introduced by a deleted statement, which avoids the redundant computation to gradually identify invalid elements.

$$\textsc{PreDel}\ \dfrac{s \in \{\textsc{New}, \textsc{Invoke}\}}{\begin{cases} \textbf{\textit{if}}\,(s \in \{\textsc{Invoke}\} \wedge \textit{k-CFA}) \vee (s \in \{\textsc{New}\} \wedge \textit{k-obj}) : \textsc{Sctx-Rule}(s) \\ \textbf{\textit{if}}\,(s \in \{\textsc{New}\} \wedge \textit{k-CFA}) \vee (s \in \{\textsc{Invoke}\} \wedge \textit{k-obj}) : \textsc{Sinv-Rule}(s) \end{cases}}$$

$$\textsc{Sctx-Rule}\ \dfrac{\begin{gathered} \begin{cases} \textit{k-CFA} : & s \in \{\textsc{Invoke}\} \\ \textit{k-obj} : & s \in \{\textsc{New}\} \end{cases} \\ s \Rightarrow c \qquad s \in m' \qquad C' \in context(m') \end{gathered}}{\begin{gathered} \forall c \in C : \begin{cases} \langle m, C\rangle \in N^-_{cg} \\ e_{cg} = \ldots \rightarrow \langle m, C\rangle : E_{cg} \setminus \{e_{cg}\}, e_{cg} \in E^-_{cg} \end{cases} \\ \forall \langle m', C'\rangle \notin N^-_{cg} : [s, \langle m', C'\rangle] \in N^*_{cg} \\ \forall e_{cg} = \langle f, [\ldots, c', c]\rangle \rightarrow \langle h, [\ldots, c']\rangle \in E_{cg} : \textsc{CheckValidity}(e_{cg}) \end{gathered}}$$

$$\textsc{Sinv-Rule}\ \dfrac{\begin{gathered} \begin{cases} \textit{k-CFA} : & s \in \{\textsc{New}\} \hookrightarrow y = new\ C() \\ \textit{k-obj} : & s \in \{\textsc{Invoke}\} \hookrightarrow x = y.g(\ldots, a_i, \ldots) \end{cases} \\ s \in m' \qquad C' \in context(m') \\ \exists e_{pag} = y \overset{invoke\ g()}{\rightsquigarrow} \\ [\textit{k-CFA}] : e_{pag} \Rightarrow s' \in \{\textsc{Invoke}\} \qquad [\textit{k-obj}] : s' = s \\ \forall o \in pts(y) : m = dispatch(o, g) \\ C \in context(m) \qquad \langle m, C\rangle \notin N^-_{cg} \qquad e_{cg} = \langle m', C'\rangle \rightarrow \langle m, C\rangle \end{gathered}}{\begin{gathered} [s', \langle m', C'\rangle] \in N^*_{cg} \\ [\textit{k-CFA}] : [s, \langle m', C'\rangle] \in N^*_{cg} \qquad [\textit{k-obj}] : E_{cg} \setminus \{e_{cg}\}, e_{cg} \in E^-_{cg} \end{gathered}}$$

$$\textsc{CheckValidity}\ \dfrac{e_{cg} = n'_{cg} \rightarrow n_{cg} \qquad e_{cg} \Rightarrow s \in \{\textsc{Invoke}\}}{\begin{cases} \textbf{\textit{if}}\ \exists n''_{cg} \rightarrow n_{cg} \in E_{cg} \wedge n''_{cg} \neq n'_{cg} : [s, n'_{cg}] \in N^*_{cg} \\ \textbf{\textit{otherwise}}\ \ n_{cg} \in N^-_{cg} \quad E_{cg} \setminus \{e_{cg}\} \quad e_{cg} \in E^-_{cg} \end{cases}}$$

$$\textsc{CheckNewSctx}\ \dfrac{e^-_{cg} = n'_{cg} \rightarrow n_{cg} \in E^-_{cg} \qquad n_{cg} = \langle f, C\rangle}{\forall s \in f : \begin{cases} \textit{k-CFA} : & s \in \{\textsc{Invoke}\} : \textsc{Sctx-Rule}(s) \\ \textit{k-obj} : & s \in \{\textsc{New}\} : \textsc{Sctx-Rule}(s) \end{cases}}$$

$$\textsc{CheckNewSinv}\ \dfrac{\forall s \in n^-_{cg} \in N^-_{cg} \qquad \begin{cases} \textit{k-CFA} : & s \in \{\textsc{New}\} \\ \textit{k-obj} : & s \in \{\textsc{Invoke}\} \end{cases}}{\textsc{Sinv-Rule}(s)}$$

Fig. 8. SHARP: PreDel: The inference rules to precompute the impact on CG = (N_{cg}, E_{cg}) after deleting Sctx and Sinv from their enclosing method m' with context C'. We overload the $\Rightarrow$ operator to more operations: $s \Rightarrow c$ means s determines a context element c, and $e_{cg} \Rightarrow s$ means e_{cg} is introduced by statement s. The red square brackets ([]) means that the premises or conclusions are limited to the specific context-sensitive algorithm indicated by the brackets.

3.3 Precompute for Deletion

The inference rules to precompute the impact on CG from a deleted Sctx and Sinv are shown in Figure 8: N_{cg}^{-} and E_{cg}^{-} represent two sets of CG nodes and edges that are no longer valid due to the deletion, and N_{cg}^{*} represents all points-to constraints originated from a statement should be deleted from its enclosing CG node. Next, we illustrate how the rules work for *k-CFA* and *k-obj*, respectively.

3.3.1 k-CFA.. **PreDel** identifies the type of input statement s and selects the correct precompute rules (*i.e.*, Sctx-Rule or Sinv-Rule) for different contexts (*i.e.*, *k-CFA* or *k-obj*).

Sctx-Rule is applied when an Invoke statement s has been removed from its enclosing method m'. Sctx determines a context element c (*i.e.*, call site). $context(m')$ maintains a set of the contexts that have arisen at call sites of each method m'. According to the two properties, we conclude that N_{cg}^{-} captures all the invalid CG nodes $\langle m, C\rangle$ introduced by the invalid context element c (*i.e.*, $c \in C$). Then we delete all incoming CG edges of those invalid CG nodes from E_{cg}, and add them to E_{cg}^{-}. Besides, all points-to constraints generated by s should be deleted from its enclosing CG node $\langle m', C'\rangle$, which we store it in N_{cg}^{*} and handle them all at the end. To avoid redundant propagation and guarantee soundness for such deletions, we apply CheckValidity rule to each CG edge of which caller has c as the last context element.

Sinv-Rule takes a deleted New statement s from method m' as the input, where s is of shape `y = new C()` (denoted by $\hookrightarrow$). According to the rule DeletePointsTo, the object node originally created by s becomes invalid and should be removed from $pts(y)$, which may further introduce more invalid points-to constraints if there exists an Invoke statement s' using y as the base variable. To guarantee the correctness, we collect the dynamic edges $y \overset{invoke\ g()}{\rightsquigarrow}$ introduced by such s' statements. Then we check whether its dispatched callee CG node $\langle m, C\rangle$ is valid (*i.e.*, $\notin N_{cg}^{-}$). If so, we add s' with $\langle m', C'\rangle$ to N_{cg}^{*} in order to remove its points-to constraints. Here, we keep the call edge e_{cg} because statement s' still exist in the program.

CheckValidity checks whether a callee node n_{cg} from the input CG edge e_{cg} is still valid, where e_{cg} is introduced by an Invoke statement s. n_{cg} is still valid when there exists any other CG edge that points to n_{cg} (except for e_{cg}). Only the points-to constraints introduced by e_{cg} is invalid and should be deleted, so that we store s with its enclosing CG node n'_{cg} to N_{cg}^{*}. Otherwise, n_{cg} and e_{cg} are invalid.

After applying all the above three rules to a deleted New or Invoke statement, we apply **CheckNewSctx** and **CheckNewSinv** to the newly concluded E_{cg}^{-} and N_{cg}^{-} in order to further collect invalid context elements and points-to constraints.

CheckNewSctx goes through each invalid CG edges $e_{cg}^{-} \in E_{cg}^{-}$, and collect the Invoke statement s from the destination node (*i.e.*, n_{cg}). This s introduces invalid CG edge(s) that should not be reached in the current program. If so, s also introduces a new invalid context element for *k-CFA*. To further discover invalid graph elements, we apply Sctx-Rule to s.

CheckNewSinv is called after running the above four rules and reaching a fixed point. This rule discovers the invalid object nodes introduced by New statements from N_{cg}^{-}: they should be treated as statement deletions and Sinv-Rule should be applied to them.

3.3.2 k-obj. This section introduces the differences when applying the rules in Figure 8 to *k-obj*.

Sctx-Rule applies when deleting a New statement, where c as a deleted allocation site becomes invalid. Similarly, we confirm the invalid elements introduced by c: CG nodes $\langle m, C\rangle$ and their incoming CG edges. Meanwhile, the introduced constraints by s should be deleted from its enclosing CG node by adding them to N_{cg}^{*}. Finally, we check the validity of the callee nodes of which c is the last context element.

$$\textsc{DeleteStmt}\ \frac{s}{\begin{cases} \boldsymbol{if}\, s \in \{\textsc{Assign}, \textsc{Load}, \textsc{Store}\} : \textsc{PAGOnly}(s) \\ \boldsymbol{if}\, s \in \{\textsc{New}, \textsc{Invoke}\} : \textsc{PAGandCG}(s) \end{cases}}$$

$$\textsc{PAGOnly}\ \frac{s \in \{\textsc{Assign}, \textsc{Load}, \textsc{Store}\} \qquad C \in context(m) \qquad \mathcal{E}_{pag} = extract(s, C)}{\forall e_{pag} \in \mathcal{E}_{pag} : \qquad \textsc{DeletePAGEdge}(e_{pag})}$$

$$\textsc{PAGandCG}\ \frac{s \in \{\textsc{New}, \textsc{Invoke}\}}{\begin{array}{c} N^*_{cg}, N^-_{cg} = \textsc{PreDel}(s) \\ \forall n^-_{cg} \in N^-_{cg}\text{: } \textsc{DeleteCGNode}(n^-_{cg}) \\ \forall [s^*, n^*_{cg}] \in N^*_{cg}\text{: } \textsc{DeleteConstraint}(s^*, n^*_{cg}) \end{array}}$$

$$\textsc{DeleteCGNode}\ \frac{n^-_{cg} = \langle m, C\rangle \qquad \forall s \in m : \mathcal{E}^-_{pag} = \mathcal{E}^-_{pag} \cup extract(s, C)}{\begin{array}{c} N_{cg} \setminus \{n^-_{cg}\} \qquad E_{pag} \setminus \mathcal{E}^-_{pag} \\ \forall e^-_{pag} \in \mathcal{E}^-_{pag} : \textsc{PropagateOrReset}(e^-_{pag}, \mathcal{E}^-_{pag}) \end{array}}$$

$$\textsc{PropagateOrReset}\ \frac{e^-_{pag} = v \longrightarrow y \vee v \rightsquigarrow y \in \mathcal{E}^-_{pag}}{\begin{array}{c} \forall e_{pag} = y \longrightarrow a \vee y \rightsquigarrow a \vee v \longrightarrow a \vee v \rightsquigarrow a : \\ \boldsymbol{if}\ e_{pag} \notin \mathcal{E}^-_{pag} \wedge e_{pag} \in E_{pag} : \textsc{DeletePAGEdge}(e_{pag}) \\ N_{pag} = N_{pag} \setminus \{v, y\} \end{array}}$$

$$\textsc{DeleteConstraint}\ \frac{s^* \in m \qquad n^*_{cg} = \langle m, C\rangle \qquad \mathcal{E}^*_{pag} = extract(s^*, C)}{\forall e^*_{pag} \in \mathcal{E}^*_{pag} : \textsc{DeletePAGEdge}(e^*_{pag})}$$

Fig. 9. SHARP: DeleteStmt: The inference rules for a deleted statement s in method m under *k-CFA* and *k-obj*.

Sinv-Rule is called to handle a deleted Invoke statement s in method m', where s is of shape `x = y.g(..., ` a_i`, ...)`. There are two differences from the rule applied to *k-CFA*: (1) s and s' are the same Invoke statement which introduce the dynamic edge; (2) we delete the introduced call edge e_{cg} since the call relation is no longer available after this deletion.

CheckValidity has no difference comparing with the one applying for *k-CFA*. Afterwards, we call the rules **CheckNewSctx** and **CheckNewSinv**.

CheckNewSctx tries to find a New statement s from method f, which instead introduces a new invalid context element (*i.e.*, allocation site). Similarly, this allocation site from s is unreachable in the program for *k-obj*, and we apply Sctx-Rule to s.

CheckNewSinv finds the invalid calls introduced by Invoke statements from N^-_{cg} and Sinv-Rule should be applied to them.

Finally, after applying all the above rules, N^*_{cg} and N^-_{cg} include all invalid points-to constraints and CG nodes. Instead of iteratively discover invalid program elements, we utilize the properties to precompute all of them without conducting any change propagation.

3.4 Formulations for Deletion

Figure 9 shows the inference rules to incrementally update the PAG and CG for a deleted statement s when considering contexts.

DeleteStmt takes an deleted statement s as input, and matches the type of s with the two rules PAGOnly and PAGandCG.

PAGOnly handles a deleted statement of type Assign, Load and Store. Because they only introduce changes in the PAG before we perform any change propagation. *extract(s, C)* is a function to extract PAG edges for a statement s under context C according to Table 1.

PAGandCG is designed to handle deletions of type New and Invoke by utilizing the impact on CG precomputed by the rules in Figure 8. Note that we firstly apply DeleteCGNode to each invalid CG node $n^{-}_{cg} \in N^{-}_{cg}$, then we handle the constraint deletions from $[s^{*}, n^{*}_{cg}] \in N^{*}_{cg}$ by DeleteConstraint.

DeleteCGNode is straightforward: it removes the input CG Node n^{-}_{cg} and extracts the PAG edges ($\mathcal{E}^{-}_{pag}$) generated by its enclosing statements under the context C. Then we call PropagateOrReset to propagate necessary points-to set changes. For the incoming and outgoing CG edges of n^{-}_{cg}, they should be already removed from CG after running Sctx-Rule and Sinv-Rule.

PropagateOrReset is designed to avoid redundant change propagation when deleting the PAG edges created from invalid CG nodes. If the two nodes v and y connected by an invalid edge e^{-}_{pag} have any outgoing PAG edges e_{pag} that are still valid (*i.e.*, $\notin \mathcal{E}^{-}_{pag} \wedge \in E_{pag}$), we apply DeletePAGEdge to those edges to propagate the changes $pts(v)$ or $pts(y)$. Finally, we simply remove v and y from PAG.

DeleteConstraint extracts the PAG edges introduced by statement s^{*} from method m under context C. All the extracted PAG edges $\mathcal{E}^{*}_{pag}$ should be removed, and we apply the rules in Figure 2 to each e^{*}_{pag} according to its type.

3.5 Illustration of Sharp

We use the two deletions ① and ② in Figure 1 to illustrate the inference rules in Figure 8 and 9 when applying them for 2-CFA.

- **Delete ① `f = new B()`:** Because the deleted statement is of type New with the context 2-CFA, DeleteStmt concludes that PAGandCG should be used and calls PreDel. Then PreDel determines Sinv-Rule is the correct precompute rule.

  Sinv-Rule firstly collects all method calls invoked by the object field variable f under its possible contexts, *i.e.*, calls using the base objects $\langle o5, [S1]\rangle$ (from $pts(o1.f)$) and $\langle o5, [S2]\rangle$ (from $pts(o3.f)$) as shown in Figure 6(a). Since there is no such call in the PAG, the concluded N^{-}_{cg} and E^{-}_{cg} are empty, which means no need to use the rules of CheckNewSctx and CheckNewSinv and hence no updated in the CG. We add two mappings to N^{*}_{cg}: the deleted statement with its two enclosing CG nodes, *i.e.*, [①, $\langle m1, [S1]\rangle$] and [①, $\langle m1, [S2]\rangle$].

  Continuing with PAGandCG, we skip DeleteCGNode because of the empty return value N^{-}_{cg}. Then DeleteConstraint is applied to N^{*}_{cg}: we extract the two PAG edges, $\langle o5, [S1]\rangle \longrightarrow \langle o1.f\rangle$ and $\langle o5, [S2]\rangle \longrightarrow \langle o3.f\rangle$, and perform DeletePAGEdge on the extracted edges as shown in Figure 2.

- **Delete ② `m2(f)`:** Sctx-Rule is used here for an deleted Invoke statement. As shown in Figure 1, ② determines the context element $S3$ in its enclosing method $m1$ (*i.e.*, $s \Rightarrow S3$ and $s \in m1$). Hence, we can conclude that all the CG nodes with $S3$ as its context element are invalid, *i.e.*, $\langle m2, [S3, S1]\rangle$, $\langle m2, [S3, S2]\rangle$ and $\langle m3, [S4, S3]\rangle$ should be included in N^{-}_{cg}. As shown by Figure 5(d), all the CG edges that point to the nodes in N^{-}_{cg} are also invalid, which are excluded from CG and included into E^{-}_{cg}. To guarantee the soundness, [②, $\langle m1, [S1]\rangle$] and [②, $\langle m1, [S2]\rangle$] are added to N^{*}_{cg}.

Afterwards, we apply the rule CheckValidity to any CG edge that has $S3$ as the last context element in its caller node, *i.e.*, $\langle m3, [S4, S3]\rangle \twoheadrightarrow \langle m4, [S5, S4]\rangle$. Since there is no other valid CG edge pointing to $\langle m4, [S5, S4]\rangle$, this node should be included to N^{-}_{cg} and the checked call edge is removed from CG and added to E^{-}_{cg}.

Then CheckNewSctx and CheckNewSinv are used on E^{-}_{cg} and N^{-}_{cg}, respectively. CheckNewSctx collects all Invoke statements in the callee node of a call edge in E^{-}_{cg} (*i.e.*, here the callee nodes are $\langle m2, [S3, S1]\rangle$, $\langle m2, [S3, S2]\rangle$, $\langle m3, [S4, S3]\rangle$ and $\langle m4, [S5, S4]\rangle$, which are the same as N^{-}_{cg}) and applies Sctx-Rule to them. CheckNewSinv collects all New statements enclosed in the nodes from N^{-}_{cg} and applies Sinv-Rule to them. After this round of running Sctx-Rule and Sinv-Rule, we found out there is no newly concluded E^{-}_{cg} and N^{-}_{cg}, and exit PreDel.

After returning to PAGandCG, we apply DeleteCGNode to N^{-}_{cg} by extracting the PAG edges (*i.e.*, $\langle p2, [S3, S1]\rangle \rightarrow \langle p3, [S4, S3]\rangle$, $\langle p2, [S3, S2]\rangle \rightarrow \langle p3, [S4, S3]\rangle$ and $\langle p3, [S4, S3]\rangle \rightarrow \langle p4, [S5, S4]\rangle$) introduced by statements in N^{-}_{cg} and applying PropagateOrReset to them. PropagateOrReset resets the points-to sets of all the pointers from the extracted PAG edges, since the pointers are all invalid.

Finally, we perform DeleteConstraint for the PAG edges (*i.e.*, $o1.f \rightarrow \langle p2, [S3, S1]\rangle$ and $o1.f \rightarrow \langle p2, [S3, S2]\rangle$) introduced by N^{*}_{cg}.

3.6 Handling Addition

Handling additions of Sctx and Sinv are different from handling their deletions, because we know that all the elements exist in the graphs before any deletion. But an addition creates new elements from none. We cannot precompute all the descendant callee methods with new contexts before confirming their existences by analyzing the pointers of their base variables after the propagation of change in the PAG. Hence, we follow the on-the-fly pointer analysis to iteratively update changes for both graphs.

3.7 The End-to-End Algorithm

This section introduces the end-to-end algorithm of Sharp after running the initial pointer analysis for the whole program. Algorithm 1 takes each code commit as an input to compute the impact on the PAG and CG from the code changes.

We firstly do git checkout for the new commit hash *commit* and rebuild the project. We also initialize two empty sets, D and A, to record all the deleted and added statements from this commit, separately. Then we utilize the git diff to obtain a set of files $java^{*}$ with code modifications (marked by GitHub with `DiffEntry.ChangeType.MODIFY`). Here, we do not consider the set of added and deleted java files, $java^{+}$ and $java^{-}$ (marked by GitHub with `DiffEntry.ChangeType.ADD` and `DiffEntry.ChangeType.DELETE`), because they might not be reachable from the analyzed main method. However, we will not miss any new code or over-consider deleted code introduced by the code commit. Because our algorithm is an on-the-fly algorithm: by starting from the modified methods introduced by the modified files, Sharp can gradually and iteratively propagate the change to reach those code in the new or deleted files.

For each modified java file f, we retrieve a set of its enclosing methods $\{method^{*}\}$. For each such method m^{*}, *CompareIRDiff* computes the new IR for the new commit and compares with its old IR from *prevCommit* to obtain the added and deleted statements. Next, we iterate each

Algorithm 1: The End-to-End Sharp.

```
Input       : $commit$ - a new git commit hash.
GlobalState : Prog - the analyzed program,
              CG - the call graph,
              PAG - the pointer assignment graph,
              $prevCommit$ - the first parent of $commit$.
1  GitCheckOut($commit$) // run git checkout to obtain this commit
2  Build(Prog) // rebuild the program with the new code changes from $commit$
3  $D \leftarrow \emptyset, A \leftarrow \emptyset$ // initialize
4  $\langle java^-, java^+, java^* \rangle \leftarrow$ GitDiff($prevCommit$, $commit$)
5  foreach $f \in java^*$ do
6      $\{method^*\} \leftarrow$ GetEnclosingMethods($f$)
7      foreach $m^* \in \{method^*\}$ do
8          $\langle \{stmt^-\}, \{stmt^+\} \rangle \leftarrow$ CompareIRDiff($m^*$) // compare the new and old IRs
9          $D = D \cup \{stmt^-\}$
10         $A = A \cup \{stmt^+\}$
11     end
12 end
13 foreach $s^- \in D$ do
14     DeleteStmt($s^-$) // handle deletions according to Figure 9
15 end
16 foreach $s^+ \in A$ do
17     AddStmt($s^+$) // handle additions according to Table 1
18 end
19 $prevCommit \leftarrow commit$
```

delete statement in D and apply the rule DeleteStmt to them. Finally, we handle the additions by following the Andersen's algorithm in Table 1 with context selections.

4 PARALLEL ALGORITHMS

4.1 Context Level

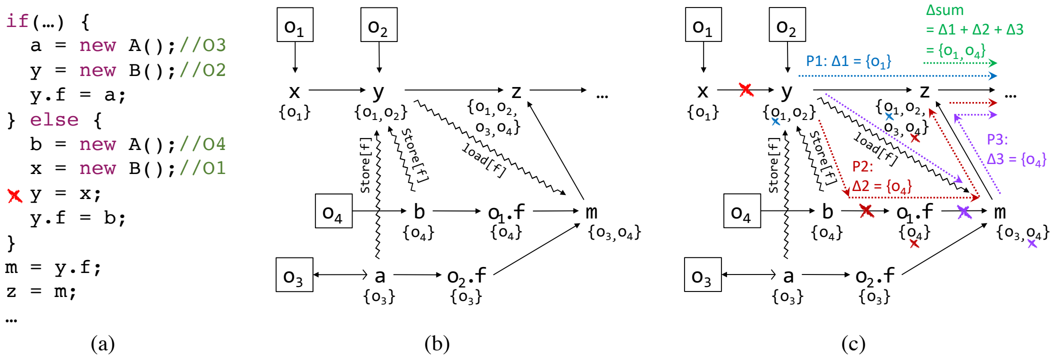
We leverage the third property to guide our parallel algorithm to incrementally update context-sensitive pointer analysis:

- Convergence: a set of invalid/newly added CG nodes are introduced by a deleted/added Sctx, which determines context element c; their context elements will gradually be replaced by existing context elements; these CG nodes will finally converge at a specific CG node(s) in CG, of which context C does not include c any more.

DeleteCGNode and its conclusion PropagateOrReset can be executed in parallel for different invalid CG nodes introduced by the same New or Invoke statement.

4.2 Code Change Level

We first discuss different scenarios of parallelization upon incremental code changes with respect to redundancy, efficiency and conflict. We then present our parallel algorithm that works for both context-sensitive and -insensitive incremental pointer analysis.



```
if(…) {
  a = new A();//O3
  y = new B();//O2
  y.f = a;
} else {
  b = new A();//O4
  x = new B();//O1
✗ y = x;
  y.f = b;
}
m = y.f;
z = m;
…
```

Fig. 10. An example to explain the intraprocedural parallelization. (a) Delete y = x from the example code. (b) The PAG before the deletion. (c) The three change propagation paths after the deletion.

4.2.1 *Intraprocedural Changes.* There are two parallel scenarios for the statement changes within a method: propagate points-to set changes in parallel for one changed statement (Section 4.2.1) and for multiple changed statements (Section 4.2.1).

One Changed Statement. Consider the example in Figure 10. Suppose we delete statement y = x from the code in Figure 10(a), which leads to the deletion of edge $x \longrightarrow y$. As shown in Figure 10(c), we run our incremental algorithm that deletes the edge $x \longrightarrow y$ with $\Delta = pts(x) = \{o_1\}$ and removes Δ from $pts(y)$. Now, three tasks of running the PROPAGATE rule for y will be performed sequentially in the following order:

Task 1: Propagate the change $\Delta 1 = pts(y) = \{o_1\}$ to z and its reachable nodes along the path $P1$ (the blue path).

Task 2: Propagation for the dynamic edge $b \overset{store[f]}{\rightsquigarrow} y$ along the path $P2$ (the maroon path): according to DELETESTORE, the edge $b \longrightarrow o_1.f$ is extracted for $\Delta 1 = pts(y) = \{o_1\}$ and should be deleted, which initializes the change $\Delta 2 = pts(b) = \{o_4\}$. Then CHECK is applied to see whether any existing incoming neighbor of $o_1.f$ has o_4 in its point-to set. Since there is no such node, the change $\Delta 2$ is confirmed and removed from $pts(o_1.f)$. Later, $\Delta 2$ is propagated to m, z and its reachable nodes, consecutively.

If there exists another path $b \longrightarrow p \longrightarrow o_1.f$, the CHECK rule will confirm that $pts(p)$ contains o_4 as an incoming neighbour of $o_1.f$. Hence, it conclude that o_4 should remain in $pts(o_1.f)$ and no change needs to be propagated.

Task 3: Propagation for the dynamic edge $y \overset{load[f]}{\rightsquigarrow} m$ along the path $P3$ (the purple path): similar to Task 2, DELETELOAD extracts edge $o_1.f \longrightarrow m$ with the initialized change $\Delta 3 = pts(o_1.f) = \{o_4\}$. Then $\Delta 3 = \{o_4\}$ is confirmed by CHECK, which is removed from $pts(m)$ and propagated to z and its reachable nodes.

Task 2 and 3 may exchange their orders, but this does not affect the final result: no matter which task is performed at first, o_4 is removed from $pts(m)$; when the other task starts to propagate the same change $\{o_4\}$, CHECK will confirm that there is no change to propagate and the propagation stops here. This example shows the worst case that all the changes ($\Delta 1$, $\Delta 2$ and $\Delta 3$) repetitively update a sequence of nodes with the summarized change Δ_{sum}, *i.e.*, z and its reachable nodes are updated by $\Delta 1$ and $\Delta 2$ twice, where $\Delta_{sum} = \Delta 1 + \Delta 2 + \Delta 3 = \{o_1, o_4\}$. For the propagation of $\Delta 3$, we can find that $\Delta 3$ has already been removed from $pts(z)$ and no change needs to be propagated. This may reduce the performance significantly.

In summary, there are three cases of the change Δ_i ($i \in [1, n]$) along their propagation paths P_i and the summarized change Δ_{sum} along the common path P_{sum} on a PAG:

Case (i) $P_{sum} = \emptyset$,
Case (ii) $P_{sum} \neq \emptyset \ \wedge \ \Delta_{sum} = \Delta_i \neq \emptyset \ \wedge \ \Delta_j = \emptyset$ ($\forall j \in [1, n]$ but $j \neq i$),
Case (iii) $P_{sum} \neq \emptyset \ \wedge \ \Delta_{sum} = \bigcup_{i=1}^{k} \Delta_i \ \wedge \ \Delta_i \neq \emptyset$ ($1 < k \leq n$).

Case (i) and (ii) have no repetitive propagation, since they either have no common propagation path or only require one time of propagation for one change. Hence, the propagations for different Δ_i can be run in parallel without conflict.

Case (iii) involves repetitive propagation as shown in the example. However, the change-consistency property from PIPA can guarantee the correctness of points-to sets if we update them in parallel. We prove this in two situations for any Δ_i and Δ_j from path P_i and P_j ($i, j \in [1, k]$):

Situation 1 If $\Delta_i \bigcap \Delta_j = \emptyset$, Check and Propagate run on P_i and P_j and remove different objects from the same points-to set, which has no conflict.

Situation 2 If $\Delta_i \bigcap \Delta_j = \delta \neq \emptyset$, no matter which path is scheduled to propagate δ first, δ will be removed from the first points-to set on P_{sum} (denoted pts_1). The later propagation from the other path will confirm that δ no longer exists in pts_1, and runs Check and Propagate for the rest change. To be specific, if P_j firstly updates pts_1, $\Delta_i^{rest} = \Delta_i \setminus \delta$; or if P_i goes first, $\Delta_j^{rest} = \Delta_j \setminus \delta$. Moreover, the two propagations after pts_1 are equivalent to Situation 1, since $\Delta_i^{rest} \bigcap \Delta_j = \Delta_i \bigcap \Delta_j^{rest} = \emptyset$.

Continue with the previous example. If we run the three tasks in parallel, Figure 10(c) explains for case (iii):

- $P1$ and $P2$ illustrate Situation 1: $\Delta 1 \bigcap \Delta 2 = \{o_1\} \bigcap \{o_4\} = \emptyset$, no matter which path propagates its change first, the finalized change in $pts(z)$ is always the same.
- $P2$ and $P3$ illustrate Situation 2: $\Delta 2 \bigcap \Delta 3 = \{o_4\} \bigcap \{o_4\} = \{o_4\} \neq \emptyset$, both paths propagates the same change to and finalized at m. For the propagation after m, we have $\delta = \{o_4\}$ and Δ_2^{rest} (if $P3$ is executed firstly) $= \Delta_3^{rest}$ (if $P2$ is executed firstly) $= \emptyset$.

According to the above reasoning, we can fully parallelize the propagation of changes caused by one statement change, which includes all the outgoing neighbors from points-to and dynamic edges in the PAG.

Multiple Statement Changes. We use an SSA-based IR [Sridharan et al. 2013] for pointer analysis, where each variable in a method are represented by a unique value number (*e.g.*, v1, v2, v3). After statement changes, IR will be recomputed by assigning new value numbers to variables in the new code. Therefore, a value number represents different variables in the source code before and after the change, which requires a complex procedure to match the new and old values for a variable.

Additionally, the pointers with large value numbers always have points-to constraints with the ones with small value numbers. Handling multiple statements from a method in parallel causes enormous useless work, since the points-to sets of large value number pointers cannot be finalized until the change propagation stemmed from small value number pointers completes.

In summary, this parallel strategy introduces redundancy. A practical solution is to sequentially compute the effect of each changed statement according to their order in a method.

4.2.2 *Interprocedural Changes.*

This is the most common scenario in our studied code commits: developers change several statements in different methods. According to the end-to-end incremental algorithm [Liu and Huang 2018; Liu et al. 2019], it separates the code changes into two sets: statement deletions and additions. They firstly solve all the deletions and then the additions. Here, we follow the same procedure.

One Changed Statement in Each Method. Assume there is only one changed statement for each method, and all the changes from different methods are either deletions or additions. Similar to Section 4.2.1, there are three cases for a change Δ_i^m along its propagation path P_i^m ($i \in [1, n]$) originated from a changed statement in method $m \in \mathbf{M}$, where $\mathbf{M}$ is a set of different methods with the same type of statement changes. Here also exists a sequence of nodes repetitively updated by some paths with change Δ_{share} along the common path P_{share}:

Case (i) $P_{share} = \emptyset$,
Case (ii) $P_{share} \neq \emptyset$ and $\Delta_{share} = \Delta_k^m \neq \emptyset$ and $\Delta_i^m = \emptyset$ ($\forall i \in [1, n]$ but $i \neq k$, $m, m \in \mathbf{M}$),
Case (iii) $P_{share} \neq \emptyset$ and $\Delta_{share} = \bigcup_{i=1}^{k} \Delta_i^m$ and $\Delta_i^m \neq \emptyset$ ($1 < k \leq n$, $m \in \mathbf{M}$).

The three cases are equivalent to the ones in Section 4.2.1, so the same proof can be easily adapted to the three cases here to conclude that: the computation of changed statements (either deletion or addition) from multiple methods (one statement from one method) can be parallel without conflict.

Multiple Changed Statements in Each Method. We assume multiple methods have changes, and each method includes several statement changes that are either deletions or additions. As we discussed in Section 4.2.1, there exist useless computations if we handle multiple statement changes of one method in parallel. Thus, applying a massive parallelization here will introduce more redundancy. Instead, we should handle the statement changes in each method sequentially.

Statement Changes with Both Deletion and Addition. We assume multiple methods have changes, each method has only one changed statement which can be deletion or addition. It is obvious that an addition can invalidate a deletion effect, and vice versa. To avoid such invalidation, an optimized schedule of processing statements is compulsory [Saha and Ramakrishnan 2006], which may or may not gain efficiency over the computation of scheduling. Hence, such a parallel strategy is unwise and we do not adopt it.

5 EVALUATION

We implemented SHARP in the WALA framework [WALA 2017] by utilizing its existing k-CFA implementation [Sridharan et al. 2013], and implemented k-obj according to the original paper [Milanova et al. 2005]. Our implementation is open-source [1]. We performed an empirical evaluation on four large, real-world Java projects on GitHub with frequent code commits. Our evaluation aims to answer the following research questions:

(1) Is SHARP as efficient as IPA?
(2) Is our parallel algorithm more efficient than the existing parallel algorithm [Liu and Huang 2018; Liu et al. 2019]?

(3) Is SHARP practical in the real-world scenario, *e.g.*, run an incremental application for code commits?

Benchmarks and Code Commits We selected the Java projects as shown in Table 2, all of which are real-world projects that are widely used and have frequent code updates in their GitHub repositories. We perform our evaluation on the most recent 10 git commits (at the date of writing) from the benchmarks' official git repositories in order to evaluate SHARP on the real-world version

[1]https://github.com/april1989/Incremental_Points_to_Analysis

Table 2. GitHub Information (for the 10 git commits).

Project	**#ΔStmt**	**#ΔNew**	**#ΔInvoke**
Hbase	+2490/-2197	-98	-839
Lucene	+2234/-530	-9	-63
Yarn	+627/-631	-36	-243
Zookeeper	+193/-242	-24	-109

control system. Here, the code update refers to source code change in *.java* files, and we ignore the git commits that changes non-Java files, *e.g.*, the updates on build files, text files or comments.

Table 2 provides the statistic information for the evaluated 10 git commits for each benchmarks. Column 2 shows the total number of added and deleted statements (denoted + and −, respectively). Columns 3 and 4 show the total number of deleted New and Invoke statements from the evaluated commits, on which we apply Sctx-Rule and Sinv-Rule (Figure 8).

Table 3 shows the statistics of our evaluated PAGs under different pointer analysis algorithms: Columns 5-7 report the number of pointers, objects and points-to edges in the PAG for each benchmark. All the benchmarks generate very large context-sensitive PAGs, which include from at least 2 millions to at most 1 billions points-to edges.

Methodology The evaluation procedure for each technique on each benchmark is as following: (1) we firstly perform the initial whole program pointer analysis on the most recent code available in GitHub (at the date of writing). Then we utilize the computed points-to analysis result (*i.e.*, PAG and CG) to initialize Sharp and apply its incremental algorithms to analyze each commit. (2) We do git checkout for each commit, rebuild the project, and utilize the git diff to compute the changed classes and methods. (3) We compute the IR between the previous and current commits to obtain a set of added and deleted statements for each changed method. Finally, we run the technique on the two sets of statements.

To compare the performance of incremental algorithms, we run Sharp on the following context-sensitive pointer analysis algorithms: 1-CFA, 2-CFA, 1-obj and 2-obj. Meanwhile, we run IPA on context-insensitive algorithm by following the above procedure.

To compare the performance of parallelization, we evaluate our parallel algorithm (denoted *Sharp (Max)*) against the existing parallel algorithm [Liu and Huang 2018; Liu et al. 2019] (denoted *Sharp (PIPA)*), both of which are running on 1-CFA, 2-CFA, 1-obj and 2-obj with 8 threads.

To make sure the whole program pointer analysis can finish within 12 hours, we exclude certain libraries from the evaluation, *e.g.*, *java.awt.** and *java.text.**. The experiment is conducted on a Linux machine with Intel Xeon E7-4860 (Westmere-EX), 8-core, 2.26GHz and 1TB memory.

5.1 Overview of Sharp

Table 3 provides an overview on the performance of Sharp: column 3 shows the full time for running different pointer analysis algorithms (indicated by column 2) for the whole project, and column 4 shows the average end-to-end time to analyze a git commit by Sharp.

In general, Sharp(Max) only requires 18 332.64 ms on average to finish the incremental analysis for a commit, which is 186.6x (highlight by red) faster than re-running the whole program analysis. Meanwhile, IPA is on average 193.5x faster than re-running the context-insensitive algorithm, which means Sharp(Max) can achieve the similar speedup as IPA does and is as efficient as IPA. The max speedup of Sharp(Max) is 880.6x when analyzing Zookeeper under 2-CFA. This proves the efficiency of Sharp(Max): rather than spending hours to run the whole program pointer analysis,

Table 3. The Overview of SHARP and Statistics of Different Pointer Analyses.

Project	Analysis	Full Time	Avg. Time Per Commit (Time: ms)		Statistics #Pointer	#Object	#Edge
Hbase	CI	2.06 min	IPA	6494.1/18.1x	310,155	22,327	228,889,050
	1-CFA	2.65 min	SHARP(Max)	11452.94/12.9x	642,685	22,987	110,152,245
	2-CFA	3.56 h		182858.63/69.2x	4,594,120	38,607	703,547,087
	1-obj	13.26 s		716.15/17.5x	70,060	4,861	2,098,257
	2-obj	17.07 s		958.0716.8x	102,526	5,712	3,803,780
Lucene	CI	9.68 s	IPA	21.5ms/449.1x	127,596	7,418	5,265,380
	1-CFA	62.67 s	SHARP(Max)	2631.47/22.8x	405,498	14,691	79,307,176
	2-CFA	7.63 h		37555.68/731.0x	3,990,653	33,271	1,094,819,728
	1-obj	25.84 s		901.41ms/27.6x	116,581	6,420	8,515,319
	2-obj	30.88 s		311.62/98.1x	143,353	8,028	12,395,079
Yarn	CI	22.96 s	IPA	406.2ms/55.5x	145,734	11,278	18,660,434
	1-CFA	44.97 s	SHARP(Max)	2062.05/20.8x	310,889	10,196	16,734,763
	2-CFA	2.47 h		11948.81/742.8x	3,714,158	20,493	370,207,476
	1-obj	50.62 s		320.36/157.0x	152,089	7,580	11,071,426
	2-obj	52 min		18637.75/165.8x	616,034	34,472	1,091,507,725
Zookeeper	CI	10.36 s	IPA	601.3/16.2x	96,238	8,705	9,308,586
	1-CFA	26.11 s	SHARP(Max)	4495.83/4.81x	301,099	9,491	33,759,731
	2-CFA	6.35 h		15517.79/880.6x	3,925,385	26,220	1,092,490,217
	1-obj	15.17 s		1243.04/11.2x	79,795	4,297	2,817,254
	2-obj	18.58 s		1710.73/9.8x	90,276	5,331	5,295,305
Avg.		1.15 h	IPA	1783.95/193.5x			
			SHARP(Max)	18332.64/186.8x			

SHARP(Max) provides fast feedback for each commit changes within 19 s, which is a super helpful and convenient way for continuous integration apps to utilize the result of pointer analysis.

5.2 Performance of Incremental Algorithms

Table 4 lists the detailed data in the performance comparison between SHARP and IPA. Columns 4-7 report the performance for each commit: columns 4 and 5 show the average time to compute the sets of added and deleted statements extracted from a commit, columns 6 and 7 report the average and worst time to run our PREDEL rules for the deleted NEW and INVOKE statements in a commit (denoted *PreDel*). Columns 8-12 report the performance for each statement: columns 8-10 show the average time to handle an added statement, a deleted statement, and to run PREDEL for one deleted NEW or INVOKE statement, while columns 11-12 show the worst case addition and deletion time.

Here, we separate the time for running *PreDel* from the one for handling deletions to clearly show the performance advantage: the average deletion time has been significantly reduced if we do not consider the interaction between the PAG and CG introduced by NEW and INVOKE statements, *e.g.*, SHARP requires on average 1982.50 ms for running *PreDel* under 1-obj but only 154.66 ms to handle all other types of statements for a commit.

We can observe that SHARP requires less execution time on *k-obj* than *k-CFA* for most benchmarks in our evaluation, especially the time spend for running the procedure *PreDel*. This is because there are more deleted INVOKE statements (6.51x on average and at most 7.56x) in each git commit as

Table 4. The Performance of Incremental Algorithms on Different Pointer Analyses (Time: ms).

Project	Analysis		Per Commit				Per Statement				
			Avg.			Worst	Avg.			Worst	
			Add	Delete	PreDel	PreDel	Add	Delete	PreDel	Add	Delete
Hbase	IPA	CI	860.4	4536.4	-	-	3.50	20.99	-	438	1895
	Sharp	1-CFA	1384.44	243.22	39927.11	352287	5.17	1.03	397.06	1120	1676
		2-CFA	250199.00	826.38	29484.13	220082	853.55	13.19	263.25	309888	220082
		1-obj	341.39	74.38	537.28	1038	1.53	1.62	74.62	832	417
		2-obj	442.94	93.33	1494.43	2024	1.98	2.04	207.56	1982	526
Lucene	IPA	CI	4.50	9.50	-	-	0.01	0.11	-	6	35
	Sharp	1-CFA	2119.25	2311.33	8.00	28	9.49	50.47	1.11	124	104
		2-CFA	56153.25	10165.25	905.75	1613	251.36	221.95	125.80	1274	3177
		1-obj	629.33	1177.83	7.42	30	2.82	25.72	0.97	24	433
		2-obj	229.40	501.94	13.33	33	1.03	10.96	1.85	13	49
Yarn	IPA	CI	173.30	236.30	-	-	2.76	3.74	-	183	161
	Sharp	1-CFA	463.83	43.21	2593.18	3606	7.40	1.23	92.95	294	96
		2-CFA	18024.91	487.24	2506.69	19016	287.48	13.84	89.85	31488	1098
		1-obj	96.47	329.89	145.28	276	1.54	9.37	5.21	104	638
		2-obj	1969.80	16435.95	15192.80	20464	31.42	466.93	544.54	1339	2278
Zookeeper	IPA	CI	46.20	1269.20	-	-	0.56	13.64	-	54	5191
	Sharp	1-CFA	96.55	88.50	5541.40	7700	4.97	8.12	416.65	239	54
		2-CFA	30468.94	321.93	6082.50	20382	1578.70	29.53	457.33	10342	4233
		1-obj	205.66	154.66	1982.50	7382	10.66	14.19	149.06	109	314
		2-obj	440.32	174.33	2463.75	6910	22.81	15.99	185.19	559	19

shown in Table 2, and the size of PAGs generated by *k-obj* are relatively smaller than the ones by *k-CFA* for the same project.

For some cases, Sharp makes the handling of statement deletions faster than additions due to the full optimizations (*i.e.*, *PreDel* and the rules in Figure 9). For example, running Sharp on 2-CFA for Yarn requires 287.48 ms on average to handle an addition, and 51.85 ms on average (the average of 13.84 ms for deletion and 89.85 ms for *PreDel*) to solve a deletion.

The worst cases of *PreDel* require several minutes to finish, which is relatively slow. We manually inspect their corresponding code commits, which always involve a large number of invalid CG nodes. For example, Sharp requires 3.4 min to finish *PreDel* under 2-CFA when analyzing the git commit 04c3888 from Hbase. This is the largest evaluated code commit for Hbase, which includes 1807 statement additions and 1537 deletions. Moreover, this commit code has 507 deleted Invoke statements, which invalids 16429 CG nodes as well as their enclosing points-to constraints. From the perspective of the volume of changes in the PAG and CG, Sharp is still quite efficient to finish such a huge amount of deletions.

5.3 Performance of Parallel Algorithms

Table 5 shows the performance comparison between our parallel algorithm (Max) and the existing parallel algorithm (PIPA) when applying both of them to our incremental analysis Sharp for different context-sensitive pointer analyses. This table reports the same attributes as Table 4. Since PIPA does not have the same parallelization as shown by Section 4.1, we use "-" to indicate no such data.

Sharp(Max) indicates 0.6x faster on average over Sharp(PIPA), and 1.3x faster over Sharp. This proves that our parallel algorithm is more faster than the existing parallel algorithm. The

Table 5. The Performance of Parallel Algorithms on Different Pointer Analyses (Time: ms).

			Per Commit				Per Statement				
				Avg.		Worst		Avg.		Worst	
Project	**Analysis**		**Add**	**Delete**	**PreDel**	**PreDel**	**Add**	**Delete**	**PreDel**	**Add**	**Delete**
Hbase	SHARP (PIPA)	1-CFA	1194.00	222.66	-	-	4.46	1.79	-	909	1043
		2-CFA	226927.75	585.87	-	-	777.48	9.40	-	237801	17137
		1-obj	303.85	59.32	-	-	1.36	1.30	-	429	312
		2-obj	385.06	83.48	-	-	1.72	1.82	-	880	195
	SHARP (Max)	1-CFA	839.01	174.83	10439.10	140532	3.76	1.82	149.88	629	582
		2-CFA	173084.81	430.34	9343.48	181359	774.78	2.27	197.71	208341	14583
		1-obj	259.28	30.93	425.94	693	1.16	0.68	59.16	292	201
		2-obj	234.99	53.75	669.33	934.29	1.05	1.17	92.96	724	76
Lucene	SHARP (PIPA)	1-CFA	1149.75	1882.75	-	-	5.15	41.11	-	107	103
		2-CFA	46153.25	7165.25	-	-	206.59	156.45	-	974	1177
		1-obj	460.49	943.20	-	-	2.06	20.59	-	16	374
		2-obj	136.40	312.07	-	-	0.61	6.81	-	11	38
	SHARP (Max)	1-CFA	904.22	1722.43	4.82	21	4.05	37.61	0.67	73	83
		2-CFA	34554.81	2657.04	343.83	932	154.68	58.01	47.75	684	592
		1-obj	268.44	629.40	3.57	13	1.20	13.74	0.50	14	241
		2-obj	78.90	219.90	12.81	12	0.35	4.80	1.78	7	23
Yarn	SHARP (PIPA)	1-CFA	383.73	34.83	-	-	6.12	0.99	-	259	63
		2-CFA	16911.94	354.70	-	-	269.73	10.08	-	31298	683
		1-obj	57.46	228.34	-	-	0.92	6.49	-	85	115
		2-obj	1313.93	11531.20	-	-	20.96	327.59	-	1052	1805
	SHARP (Max)	1-CFA	367.35	31.33	1663.37	2485	5.86 0.89	59.62	192.33	58	
		2-CFA	10803.17	178.91	966.73	9932	172.30	5.08	34.65	11743	203
		1-obj	43.80	172.68	103.88	214	0.70	4.91	3.72	47	565
		2-obj	909.88	8993.16	8734.71	11613	14.51	255.49	313.07	739	793
Zookeeper	SHARP (PIPA)	1-CFA	92.60	48.63	-	-	4.80	4.46	-	212	39
		2-CFA	24378.80	210.00	-	-	1263.15	19.27	-	7271	57
		1-obj	93.15	29.93	-	-	4.82	2.66	-	51	211
		2-obj	348.38	168.50	-	-	18.05	15.46	-	273	17
	SHARP (Max)	1-CFA	58.46	24.76	4412.60	4923	3.03	2.27	331.77	193	34
		2-CFA	12381.60	116.32	3019.87	9342	641.53	10.67	227.06	6432	2050
		1-obj	76.94	16.63	1149.47	5934	3.99	1.53	86.43	29	110
		2-obj	300.35	116.25	1294.12	2937	15.56	10.67	97.30	172	14

actual speedup for each commits are quite different due to the distribution of code changes: more parallelization can be utilized if a commit contains multiple code changes in different methods.

In summary, SHARP works quite well on real-world projects with frequent code commits.

5.4 Discussions

5.4.1 How does k affect the performance of SHARP. The performance difference of SHARP between 2-CFA and 1-CFA is different from the one between 2-obj and 1-obj. This is affected not only by the value of k, but also by the context-sensitive algorithm (*i.e.*, use a call site or a receiver object). According to the statistics in Table 3, 2-CFA always creates much larger sizes of PAG over 1-CFA when comparing the PAG sizes created by 2-obj over 1-obj. For example, 2-CFA computes 1,094,819,728 edges for Lucene, which is about 14x more than the one computed by 1-CFA; while 2-obj only introduces 12,395,079 edges, which is around 1.5x more than the one from 1-obj. Hence, *k-CFA* essentially involves more computation over *k-obj* (when both algorithms use the same k).

Our incremental rules are based on *k-CFA* and *k-obj*, which performance follows the same trends as the corresponding context-sensitive algorithms.

When it comes to a larger k (*i.e.*, $k \geq 3$), the scalability of *k-CFA* and *k-obj* becomes even worse due to the significantly increased size of constraints. Theoretically, the worst-case complexity can be doubly exponential [Sagiv et al. 1998]. Hence, it is impractical to use $k \geq 3$ in real-world scenarios where the whole program pointer analysis probably cannot terminate. This is also why most context-sensitive pointer analysis works use $k = 1$ and $k = 2$ in their evaluations [Jeon et al. 2018; Kastrinis and Smaragdakis 2013; Li et al. 2018b; Smaragdakis et al. 2011, 2014b]. Consequently, SHARP is required to handle more constraints introduced by a large k for incremental changes, but should still be much faster than re-running the whole pointer analysis.

5.4.2 Threats to Validity. There are three threats to validity of our evaluation: the limited set and manual selection of Java programs (4) used to compare the performance between IPA and SHARP, the manual selection of main entry method for each program and the small number of GitHub commits (10) for each evaluated program. We have partially mitigated the first threat to validity by selecting the real-world programs with large code bases (*i.e.*, around 1 MLOC) and frequent GitHub commits with a long history (*i.e.*, at least 10 years). The remaining two threats to validity are due to the fact that many main methods in a project can be used as the entry point to initiate pointer analysis, however, some of which cannot terminate especially when we are evaluating context-sensitive algorithms (*i.e.*, *k-CFA* and *k-obj*). Moreover, the GitHub commits may change different parts of a project, which may not be able to touch the points-to results generated for a main method that can terminate for all the evaluated algorithms (*i.e.*, 1-CFA, 2-CFA, 1-obj and 2-obj). Hence, it is difficult to collect a large number of commits that change the points-to result for a terminatable main method. We have partially mitigated this threat by selecting a configuration that satisfies all the mentioned requirements, *i.e.*, a main method for each program that can terminate for all evaluated context-sensitive algorithms with enough GitHub commits (10) that change the points-to results.

6 RELATED WORK

Most existing incremental pointer analysis algorithms are derived from the deletion-rederivation (DRed) algorithm [Gupta et al. 1993]. For example, the first incremental flow- and context-sensitive alias analysis proposed by Yur et al. [Yur et al. 1999], and the reset-recompute algorithm from ECHO [Zhan and Huang 2016] for Java programs.

Almost all the algorithms based on DRed have redundant computation, due to the reset without checking. Saha et al. [Saha and Ramakrishnan 2005a,b] propose an incremental context-sensitive pointer analysis based on DRed and tabled evaluation, which performs a check before reset. But the check cannot fully avoid redundancy and requires extra effort to maintain a *support graph*. Moreover, it cannot solve dynamic changes of method calls precisely.

IPA [Liu and Huang 2018; Liu et al. 2019] improves DRed by observing the change-locality property in the PAG. SHARP extends IPA to consider contexts and reduces the redundancy in change propagation when deleting method calls. Besides, our approach maintains CG and PAG on-the-fly to guarantee the precision.

Lu et al. [Lu et al. 2013] present an incremental pointer analysis based on CFL-reachability, where a points-to query is answered by finding the CFL-reachable paths in the PAG from the queried variable to objects. Our approach discovers the incremental algorithm in *k-CFA*, which is another widely used context-sensitive approach focused on the whole program with higher precision.

To achieve efficiency in context-sensitivity, cloning-based technique [Whaley and Lam 2004] creates multiple instances for a method and each links with a distinct context. This can be adapted to solve incrementally added statements, however, it cannot handle code deletion which requires complicate algorithms to guarantee the soundness. Similarly, *diff-graph* [Krainz and Philippsen 2017] represents the accesses of a method in a flow-sensitive, but context-insensitive way in order to reduce the workload of creating a points-to graph [Marron 2008], which however only handles addition. SHARP develops efficient algorithms especially for incremental deleted statements, which can scale on large programs.

Most existing parallel algorithms [Edvinsson et al. 2011; Méndez-Lojo et al. 2010, 2012; Nagaraj and Govindarajan 2013; Putta and Nasre 2012; Su et al. 2014; Zhao et al. 2018] are designed to speed up the whole-program pointer analysis. IPA firstly proposes the parallel algorithm for incremental pointer analysis. As explained in Section 2, they massively parallel the change propagation along points-to edges within each iteration of the fixed-point computation. Inspired by the real-world code commits, SHARP maximizes the usage of multicore machines by handling the change propagation along both points-to and dymanic edges in parallel, as well as the changed statements from different methods in parallel.

Many recent techniques [Guyer and Lin 2003; Hassanshahi et al. 2017; Jeon et al. 2018; Jeong et al. 2017; Kastrinis and Smaragdakis 2013; Li et al. 2018a,b; Smaragdakis et al. 2014a; Wei and Ryder 2015] focus on selective context-sensitivity, which is another promising direction to scale pointer analysis. The idea is to analyze selective program elements context-sensitively to avoids the combinatorial explosion in analysis time and memory space. The selection relies on heuristics from user inputs or machine learning techniques, or pattern matching of the imprecision in dataflow, which shows significant performance and/or precision improvement. The selective context-sensitive techniques and SHARP are both built on *k-CFA* and *k-obj*. However, these two types of techniques have different focuses: context selection is designed to analyze the whole program, while SHARP leverages small code changes to incrementally and soundly update the points-to results to maintain a high precision.

7 CONCLUSION

We have presented SHARP, an efficient incremental algorithm for context-sensitive pointer analysis. SHARP addresses deep technical challenges in the state-of-the-art incremental but context-insensitive pointer analysis, such as inefficient computations in handling method call deletions, soundness and parallelization. Our evaluation on real-world Java projects demonstrates that SHARP scales to real-world large complex codebases with frequent code commits, and it has even comparable performance to the state-of-the-art context-insensitive incremental pointer analysis.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback on earlier versions of this paper. This work was supported by NSF award CCF-2006450.

REFERENCES

L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, 1994.

M. A. Bender, J. T. Fineman, S. Gilbert, and R. E. Tarjan. A new approach to incremental cycle detection and related problems. *ACM Trans. Algorithms*, 12(2):14:1–14:22, Dec. 2015. ISSN 1549-6325. doi: 10.1145/2756553. URL http://doi.acm.org/10.1145/2756553.

M. Edvinsson, J. Lundberg, and W. Löwe. Parallel points-to analysis for multi-core machines. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, pages 45–54, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0241-8. doi: 10.1145/1944862.1944872. URL http://doi.acm.org/10.1145/1944862.1944872.

A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 157–166, New York, NY, USA, 1993. ACM. ISBN

0-89791-592-5. doi: 10.1145/170035.170066. URL http://doi.acm.org/10.1145/170035.170066.

S. Z. Guyer and C. Lin. Client-driven pointer analysis. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 214–236, Berlin, Heidelberg, 2003. Springer-Verlag. ISBN 3-540-40325-6. URL http://dl.acm.org/citation.cfm?id=1760267.1760284.

B. Hardekopf and C. Lin. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 290–299, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595936332. doi: 10.1145/1250734.1250767. URL https://doi.org/10.1145/1250734.1250767.

B. Hardekopf and C. Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 289–298, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-61284-356-8. URL http://dl.acm.org/citation.cfm?id=2190025.2190075.

B. Hassanshahi, R. K. Ramesh, P. Krishnan, B. Scholz, and Y. Lu. 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 2017, pages 13–18, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5072-3. doi: 10.1145/3088515.3088519. URL http://doi.acm.org/10.1145/3088515.3088519.

M. Jeon, S. Jeong, and H. Oh. Precise and scalable points-to analysis via data-driven context tunneling. *Proc. ACM Program. Lang.*, 2(OOPSLA):140:1–140:29, Oct. 2018. ISSN 2475-1421. doi: 10.1145/3276510. URL http://doi.acm.org/10.1145/3276510.

S. Jeong, M. Jeon, S. Cha, and H. Oh. Data-driven context-sensitivity for points-to analysis. *Proc. ACM Program. Lang.*, 1 (OOPSLA):100:1–100:28, Oct. 2017. ISSN 2475-1421. doi: 10.1145/3133924. URL http://doi.acm.org/10.1145/3133924.

G. Kastrinis and Y. Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 423–434, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462191. URL http://doi.acm.org/10.1145/2491956.2462191.

J. Krainz and M. Philippsen. Diff graphs for a fast incremental pointer analysis. In *Proceedings of the 12th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ICOOOLPS'17, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350884. doi: 10.1145/3098572.3098578. URL https://doi.org/10.1145/3098572.3098578.

O. Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, December 2002.

O. Lhoták and L. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1):3:1–3:53, Oct. 2008. ISSN 1049-331X. doi: 10.1145/1391984.1391987. URL http://doi.acm.org/10.1145/1391984.1391987.

Y. Li, T. Tan, A. Møller, and Y. Smaragdakis. Scalability-first pointer analysis with self-tuning context-sensitivity. ESEC/FSE 2018, page 129–140, New York, NY, USA, 2018a. Association for Computing Machinery. ISBN 9781450355735. doi: 10.1145/3236024.3236041. URL https://doi.org/10.1145/3236024.3236041.

Y. Li, T. Tan, A. Møller, and Y. Smaragdakis. Precision-guided context sensitivity for pointer analysis. 2(OOPSLA), oct 2018b. doi: 10.1145/3276511. URL https://doi.org/10.1145/3276511.

B. Liu and J. Huang. D4: Fast concurrency debugging with parallel differential analysis. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 359–373, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5698-5. doi: 10.1145/3192366.3192390. URL http://doi.acm.org/10.1145/3192366.3192390.

B. Liu, J. Huang, and L. Rauchwerger. Rethinking incremental and parallel pointer analysis. *ACM Trans. Program. Lang. Syst.*, 41(1):6:1–6:31, Mar. 2019. ISSN 0164-0925. doi: 10.1145/3293606. URL http://doi.acm.org/10.1145/3293606.

Y. Lu, L. Shang, X. Xie, and J. Xue. An incremental points-to analysis with cfl-reachability. In *Proceedings of the 22Nd International Conference on Compiler Construction*, CC'13, pages 61–81, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 978-3-642-37050-2. doi: 10.1007/978-3-642-37051-9_4. URL http://dx.doi.org/10.1007/978-3-642-37051-9_4.

M. Marron. *Modeling the Heap: A Practical Approach.* PhD thesis, USA, 2008. AAI3346744.

M. Méndez-Lojo, A. Mathew, and K. Pingali. Parallel inclusion-based points-to analysis. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 428–443, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0203-6. doi: 10.1145/1869459.1869495. URL http://doi.acm.org/10.1145/1869459.1869495.

M. Méndez-Lojo, M. Burtscher, and K. Pingali. A gpu implementation of inclusion-based points-to analysis. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 107–116, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1160-1. doi: 10.1145/2145816.2145831. URL http://doi.acm.org/10.1145/2145816.2145831.

A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, Jan. 2005. ISSN 1049-331X. doi: 10.1145/1044834.1044835. URL http://doi.acm.org/10.1145/1044834.1044835.

V. Nagaraj and R. Govindarajan. Parallel flow-sensitive pointer analysis by graph-rewriting. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 19–28, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4799-1021-2. URL http://dl.acm.org/citation.cfm?id=2523721.2523728.

S. Putta and R. Nasre. Parallel replication-based points-to analysis. In *Proceedings of the 21st International Conference on Compiler Construction*, CC'12, pages 61–80, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-28651-3. doi: 10.1007/978-3-642-28652-0_4. URL http://dx.doi.org/10.1007/978-3-642-28652-0_4.

M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst.*, 20(1):1–50, jan 1998. ISSN 0164-0925. doi: 10.1145/271510.271517. URL https://doi.org/10.1145/271510.271517.

D. Saha and C. R. Ramakrishnan. Incremental and demand-driven points-to analysis using logic programming. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '05, pages 117–128, New York, NY, USA, 2005a. ACM. ISBN 1-59593-090-6. doi: 10.1145/1069774.1069785. URL http://doi.acm.org/10.1145/1069774.1069785.

D. Saha and C. R. Ramakrishnan. Symbolic support graph: A space efficient data structure for incremental tabled evaluation. In M. Gabbrielli and G. Gupta, editors, *Logic Programming, 21st International Conference, ICLP 2005, Sitges, Spain, October 2-5, 2005, Proceedings*, volume 3668 of *Lecture Notes in Computer Science*, pages 235–249. Springer, 2005b. doi: 10.1007/11562931_19. URL https://doi.org/10.1007/11562931_19.

D. Saha and C. R. Ramakrishnan. A local algorithm for incremental evaluation of tabled logic programs. In *Proceedings of the 22nd International Conference on Logic Programming*, ICLP'06, page 56–71, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3540366350. doi: 10.1007/11799573_7. URL https://doi.org/10.1007/11799573_7.

L. Shang, Y. Lu, and J. Xue. Fast and precise points-to analysis with incremental cfl-reachability summarisation: Preliminary experience. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 270–273, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1204-2. doi: 10.1145/2351676.2351720. URL http://doi.acm.org/10.1145/2351676.2351720.

O. Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, Carnegie Mellon University, May 1991.

Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: Understanding object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 17–30, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926390. URL http://doi.acm.org/10.1145/1926385.1926390.

Y. Smaragdakis, G. Kastrinis, and G. Balatsouras. Introspective analysis: Context-sensitivity, across the board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 485–495, New York, NY, USA, 2014a. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594320. URL http://doi.acm.org/10.1145/2594291.2594320.

Y. Smaragdakis, G. Kastrinis, and G. Balatsouras. Introspective analysis: Context-sensitivity, across the board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 485–495, New York, NY, USA, 2014b. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594320. URL http://doi.acm.org/10.1145/2594291.2594320.

M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 387–400, New York, NY, USA, 2006. ACM. ISBN 1-59593-320-4. doi: 10.1145/1133981.1134027. URL http://doi.acm.org/10.1145/1133981.1134027.

M. Sridharan, D. Gopan, L. Shan, and R. Bodík. 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 '05, pages 59–76, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: 10.1145/1094811.1094817. URL http://doi.acm.org/10.1145/1094811.1094817.

M. Sridharan, S. Chandra, J. Dolby, S. J. Fink, and E. Yahav. Aliasing in object-oriented programming. chapter Alias Analysis for Object-oriented Programs, pages 196–232. Springer-Verlag, Berlin, Heidelberg, 2013. ISBN 978-3-642-36945-2. URL http://dl.acm.org/citation.cfm?id=2554511.2554523.

Y. Su, D. Ye, and J. Xue. Parallel pointer analysis with cfl-reachability. In *Proceedings of the 2014 Brazilian Conference on Intelligent Systems*, BRACIS '14, pages 451–460, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-5618-0. doi: 10.1109/ICPP.2014.54. URL http://dx.doi.org/10.1109/ICPP.2014.54.

T. Tan, Y. Li, and J. Xue. 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, pages 278–291, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4988-8. doi: 10.1145/3062341.3062360. URL http://doi.acm.org/10.1145/3062341.3062360.

WALA. T. j. watson libraries for analysis (wala). http://wala.sourceforge.net/, 2017.

S. Wei and B. G. Ryder. Adaptive context-sensitive analysis for javascript. In J. T. Boyland, editor, *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 712–734, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-86-6. doi: 10.4230/LIPIcs.ECOOP.2015.712. URL http://drops.dagstuhl.de/opus/volltexte/2015/5244.

J. Whaley and M. S. Lam. 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, pages 131–144, New York, NY, USA, 2004. ACM. ISBN 1-58113-807-5. doi: 10.1145/996841.996859. URL http://doi.acm.org/10.1145/996841.996859.

J.-s. Yur, B. G. Ryder, and W. A. Landi. An incremental flow- and context-sensitive pointer aliasing analysis. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 442–451, New York, NY, USA, 1999. ACM. ISBN 1-58113-074-0. doi: 10.1145/302405.302676. URL http://doi.acm.org/10.1145/302405.302676.

S. Zhan and J. Huang. Echo: Instantaneous in situ race detection in the ide. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 775–786, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4218-6. doi: 10.1145/2950290.2950332. URL http://doi.acm.org/10.1145/2950290.2950332.

J. Zhao, M. G. Burke, and V. Sarkar. Parallel sparse flow-sensitive points-to analysis. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, pages 59–70, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5644-2. doi: 10.1145/3178372.3179517. URL http://doi.acm.org/10.1145/3178372.3179517.