Rethinking Incremental and Parallel Pointer Analysis

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

Pointer analysis is at the heart of most interprocedural program analyses. However, scaling pointer analysis to large programs is extremely challenging. In this article, we study incremental pointer analysis and present a new algorithm for computing the points-to information incrementally (*i.e.*, upon code insertion, deletion, and modification). Underpinned by new observations of incremental pointer analysis, our algorithm significantly advances the state-of-the-art in that it avoids redundant computations and the expensive graph reachability analysis, and preserves the precision as the corresponding whole program exhaustive analysis. Moreover, it is parallel within each iteration of the fixed-point computation. We have implemented our algorithm, IPA, for Java based on the WALA framework and evaluated its performance extensively on real-world large complex applications. Experimental results show that IPA achieves more than 200X speedups over existing incremental algorithms, two to five orders of magnitude faster than the whole program pointer analysis, and also improves the performance of an incremental data race detector by orders of magnitude. Our IPA implementation is open source and has been adopted by WALA.

CCS Concepts: • **Theory of computation → Program analysis**; • **Computing methodologies → Massively parallel algorithms**;

Additional Key Words and Phrases: Incremental Pointer Analysis, Parallelization, Dynamic Graph Algorithms.

ACM Reference Format:
Bozhen Liu, Jeff Huang, and Lawrence Rauchwerger. 2018. Rethinking Incremental and Parallel Pointer Analysis. 1, 1 (November 2018), 31 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Pointer analysis, or points-to analysis, computes the set of objects that a pointer variable can point to at runtime. It is a fundamental technique underpinning virtually all interesting static program analyses (*e.g.*, compiler optimizations and bug detection) and has been the focus of intensive research [3, 15, 21, 23, 28, 32, 34, 50, 59, 65, 66, 68, 71, 73, 86].

Unfortunately, scaling pointer analysis up to large code bases has been extremely challenging. For example, for the classical Andersen's pointer analysis, it typically takes tens of minutes or hours to analyze real-world applications with hundreds of thousands of lines of code [14, 65, 80]. As a result, typically only imprecise pointer analyses are used in production compilers, missing many potential optimization opportunities [16].

Researchers have investigated incremental pointer analysis since nearly two decades ago [58, 87]. These incremental algorithms [4, 31, 45, 60, 61, 63, 79, 88] promise significant performance improvements over the exhaustive pointer analysis because analyzing code changes is often much

Authors' addresses: Bozhen Liu, Texas A&M University, College Station, TX, 77843, USA, april1989@tamu.edu; Jeff Huang, Texas A&M University, College Station, TX, 77843, USA, jeff@cse.tamu.edu; Lawrence Rauchwerger, rwerger@tamu.edu, Texas A&M University, College Station, TX, 77843, USA.

faster than analyzing the entire code base. For applications in which pointer analysis has to run repeatedly with respect to frequent but *small* program changes, *e.g.*, bug finding in the IDE (Integrated Development Environment) [88] or incremental compilation [67], this is particularly useful because only a small part of the program needs to be analyzed instead of the whole program.

However, existing incremental algorithms (albeit fast in simple cases) are still too slow to be applied in large real-world applications. For instance, in our experiments we find that existing algorithms [88] can take over half an hour to analyze a statement deletion in large applications. Moreover, most existing algorithms [4, 31, 60, 63] assume a pre-built call graph of the program, which does not hold for scenarios where the call graph itself can be modified by the code changes. Furthermore, some algorithms (e.g., [63]) do not preserve precision but compute a less precise result than the exhaustive analysis.

In this article, we perform a detailed study of the Andersen-style incremental pointer analysis, and present a new incremental algorithm that dramatically improves the performance of existing algorithms. Our algorithm does not assume a pre-built call graph and does not lose precision. More importantly, it is much more efficient than existing algorithms for handling code deletions, by exploiting a novel insight on the fundamental transitivity property of the fixed-point based pointer analysis. We show that to correctly handle a deletion, it is sufficient to analyze the *local neighbours* of the changed nodes in the pointer assignment graph (PAG) without global graph reachability analysis, nor any recomputation of the intermediate points-to results. Besides, we present an incremental algorithm that dynamically updates the strongly-connected components (SCCs) upon incremental program changes to further reduce redundant computations.

Moreover, we observe a strong *change idempotency* property of our incremental pointer analysis algorithm, which allows efficient parallelization within each iteration of the fixed-point computation. Specifically, we show that for both code insertion and deletion changes, the propagation of information along the edges of the PAG is performed using an idempotent operator, *i.e.*, the repeated update of a node in the graph with the same information (coming along various paths of the graph) does not change its state. Nor is the order of these updates important (operator is commutative, as well as associative). This property also enables us to develop a *synchronization-free* implementation of the points-to data structure, which further improves the performance of our algorithm.

We implemented an end-to-end incremental pointer analysis, IPA, in the WALA framework [85] based on our new algorithm, and evaluated it extensively on a wide range of real-world large complex Java applications from the DaCapo-9.12 benchmarks [8]. The experimental results show dramatic efficiency and scalability improvements compared to existing algorithms: on a 48-core HPC machine, IPA takes only 24ms on average and 5.5s in the worst case to analyze a change, achieving more than 200X speedups over existing incremental algorithms based on graph reachability or recomputation, and it is two to five orders of magnitude faster than the exhaustive pointer analysis while preserving the precision. We have also applied IPA to detect data races incrementally in the IDE. Through the use of IPA for incremental pointer analysis, the performance of a state-of-the-art incremental race detector [88] is improved by as much as 100X.

To our knowledge, IPA is the first parallel incremental pointer analysis that realizes both the change incrementalism and the algorithmic parallelization. Although previous research has proposed separately a number of incremental algorithms [4, 60, 79] and parallel algorithms [17, 49, 52, 55, 76, 89], none of them is both incremental and parallel, which is challenging to design and implement efficiently.

In summary, we claim the following contributions:

- We present a new pointer analysis algorithm that dramatically improves the performance and practicality of existing algorithms without losing precision. In particular, we present the first parallel incremental pointer analysis algorithm by exploiting a novel change idempotency property of incremental pointer analysis.

- We present an extensive evaluation of IPA on large complex real-world Java programs as well as an application for IDE-based incremental race detection, demonstrating significant performance improvements over existing algorithms.
- IPA is open source [30] and has been integrated into the popular WALA framework.

The remainder of the paper is organized as follows. Sections 2 and 3 introduce necessary background of pointer analysis and existing incremental algorithms; Section 4 presents our new incremental and parallel algorithms; Section 5 presents our IPA implementation and Section 7 presents the evaluation; Section 8 discusses related work and Section 9 concludes this paper.

2 ANDERSEN'S POINTER ANALYSIS

Like many other static analyses, a precise pointer analysis is undecidable [35, 56]. Andersen's analysis [3] is well known for computing the *may* points-to information using subset constraints. It is *sound* in that each computed may points-to set contains all objects the variable may point to at runtime. Another classical method is Steensgaard's analysis [75], which computes pointer analysis with a set equality constraint, but is less precise than Andersen's analysis. There are many dimensions in Andersen's analysis, and we focus on context-insensitive, flow-insensitive, and field-sensitive Andersen's points-to analysis in this paper.

```
x = new T() //o1
y = new T() //o2
y = x
x.f = y
```

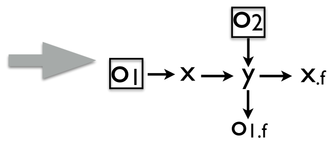

Fig. 1. An example of pointer analysis and the pointer assignment graph (PAG).

We use a program fragment of an object-oriented programing language (Java) in Figure 1 to illustrate Andersen's analysis and the key terminologies. The goal of pointer analysis is to determine the set of objects that each of the variables (*i.e.*, `x`, `y` and `x.f`) can point to. The set of objects corresponding to a variable v is called the *points-to set* of v, denoted by $pts(v)$. For example, in this program fragment, we have $pts(x) = \{o1\}$, $pts(y) = \{o_1, o_2\}$ and $pts(x.f) = \{o_1, o_2\}$.

Pointer Assignment Graph (PAG). Andersen's algorithm can be expressed as a transitive closure computation over a constraint graph, also called PAG in Lhoták's thesis [37]. We use the PAG terminology throughout this paper. The nodes in the PAG represent memory locations and program variables (including field and array accesses). The (directed) edges in the PAG represent subset relations between the points-to sets of the connected nodes. For example, in Figure 1, each program variable (`x`, `y` and `x.f`) corresponds to a node in the PAG, the edge $o_1 \longrightarrow x$ means that $o_1 \in pts(x)$, and the edge $x \longrightarrow y$ means that $pts(x) \subseteq pts(y)$. Since the analysis is field-sensitive, points-to sets both for variables, *e.g.*, $pts(x)$, and for object fields of abstract locations, *e.g.*, $pts(o_i.f)$, need to be maintained. Hence, an additional node $o_1.f$ is generated in the PAG. It is an intermediate node used in Andersen's analysis to compute the points-to constraints between field variables.

Table 1 summarizes the rules of Andersen's analysis for computing the points-to constraints corresponding to different types of program statements. For each type, the points-to sets of the related variables are updated as follows:

[Base] $x = new\ C()$: add a new object o to $pts(x)$.

Table 1. Andersen's analysis (o is an object and f is an object field).

Type	Statement	Points-to Constraints
Base	x=new T()	$pts(x) = pts(x) \cup o$
Simple	x = y	$pts(x) = pts(x) \cup pts(y)$
Field Load	x = y.f	$\forall o \in pts(y)$: $pts(x) = pts(x) \cup pts(o.f)$
Field Store	x.f = y	$\forall o \in pts(x)$: $pts(o.f) = pts(o.f) \cup pts(y)$
Array Load	x = y[i]	$\forall o \in pts(y)$: $pts(x) = pts(x) \cup pts(o.*)$
Array Store	x[i] = y	$\forall o \in pts(x)$: $pts(o.*) = pts(o.*) \cup pts(y)$

[Simple] $x = y$: add $pts(y)$ to $pts(x)$.
[Field Load*] $x = y.f$: for each object $o \in pts(y)$, add $pts(o.f)$ to $pts(x)$.
[Field Store*] $x.f = y$: for each object $o \in pts(x)$, add $pts(y)$ to $pts(o.f)$.
[Array Load]** $x = y[i]$: for each object $o \in pts(y)$, add $pts(o.*)$ to $pts(x)$.
[Array Store]** $x[i] = y$: for each object $o \in pts(x)$, add $pts(y)$ to $pts(o.*)$.
*f is an object field.
**Different array indices are not distinguished but represented by a special constant "$*$".

For interprocedural analysis, Andersen's analysis must be extended to handle an additional type of statements: *method calls*. This is straightforward when the callee method (*i.e.*, the target of the method call) can be easily identified. For example, for a method call $a = b.m(c)$, suppose a, b and c are all reference variables and m has the formal parameter p and return variable r, one can analyze the statements in m and generate two additional [simple] statements to connect the interprocedural data flows: $p = c$ and $a = r$.

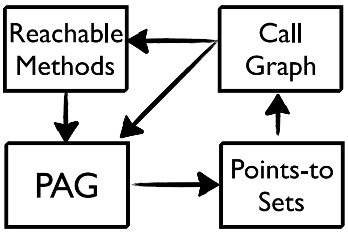

Fig. 2. Circular dependency in pointer analysis.

However, for real-world programs with dynamic dispatch, the callee method may depend on the receiver object, *e.g.*, $b.m$ depends on the object pointed by the base variable b. In other words, resolving the method calls (*a.k.a.* call graph construction [21]) relies on pointer analysis. Therefore, there is a *circular dependency* between pointer analysis and call graph construction, as illustrated in Figure 2. We refer the readers to the work of Ali and Lhoták [2] for a detailed analysis of the circular dependency.

On-the-fly Andersen's Analysis. To address this problem, Andersen's analysis has been extended to build the PAG dynamically (also called *on-the-fly*) while constructing the call graph [37, 72]. Algorithm 1 outlines the flow of the on-the-fly algorithm, which works iteratively until a *fixed point* is reached. The algorithm starts with only the initial reachable methods (*e.g.*, the main method). In each iteration, an input set of call targets are consumed and potentially a new set of points-to constraints are produced, which are again consumed by running Andersen's analysis. Specifically, each iteration consists of two steps:

ALGORITHM 1: On-the-fly Andersen's analysis

```
1 $\Delta_1 \leftarrow \emptyset$;// $\Delta_1$: the new points-to constraints in each iteration
2 $\Delta_2 \leftarrow$ initial method call targets;
3 while $\Delta_2 \neq \emptyset$ do // repeat until $\Delta_2$ is empty
4     $\Delta_1$ = extractNewMethodCallConstraints($\Delta_2$)
5     $\Delta_2$ = runAndersensAnalysis($\Delta_1$)
6 end
```

(1) Resolve new method call targets based on the current points-to information (*i.e.*, the points-to set of the receiver variable in the method call statements), and extract new points-to constraints for the newly discovered method calls.
(2) Evaluate the new constraints by following Andersen's analysis in Table 1 to compute the points-to set of each variable and discover new call targets if available.

The discovered new call targets from the second step are then provided to the next iteration as the input set. In practice, a *worklist* is often used to maintain the set of constraints.

Complex Statements. Notice that the load/store and call statements may generate more than one PAG edge, which is determined by the points-to set of the base variable (*e.g.*, `y` in `x=y.f` and `x=y[i]`). We call these statements *complex statements*. A complex statement can introduce multiple edges because its base variable may point to multiple objects. For example, for `x=y.f` if both objects o_1 and o_2 are added to $pts(y)$, then two edges $o_1.f \longrightarrow x$ and $o_2.f \longrightarrow x$ will be generated in the PAG.

SCC. An important optimization for pointer analysis in practice is to compute the *strongly-connected component*s (SCCs) in the PAG. Because all nodes in the same cycle are guaranteed to have identical points-to sets, all these nodes can be collapsed into a single node. Hence, the points-to sets of all the nodes in an SCC can be updated all together to take advantage of cycle elimination. However, existing work [23] that uses SCCs for pointer analysis only applies to non-incremental analysis. To handle dynamic code changes, SCCs must also be updated dynamically.

3 EXISTING INCREMENTAL ALGORITHMS

Incremental pointer analysis must handle two basic types of changes: *inserting* a statement and *deleting* a statement, because any code change can be composed from one or more insertions and/or deletions.

Handling insertion is straightforward based on the on-the-fly Andersen's analysis described in Algorithm 1 in the previous section. New points-to constraints can be first extracted from the inserted statement and then provided as the input to run the on-the-fly algorithm.

However, deletion is much more complicated than insertion to handle. For deletion, one has to maintain provenance information on how facts are derived. When a statement is deleted, one has to delete all facts that are "*no longer reachable*" from existing statements through the provenance information. Consider three consecutive code changes: inserting a statement `b=a`, inserting another statement `c=b`, and deleting the first statement `b=a`. When `b=a` is inserted, $pts(b)$ is updated to $pts(b) \cup pts(a)$. When `c=b` is inserted, similarly, $pts(c)$ is updated to $pts(c) \cup pts(b)$. However, when `b=a` is deleted, not only the change in $pts(b)$ should be reversed, but also that the change in $pts(c)$ should be *recomputed*, because $pts(c)$ was previously updated based on $pts(b)$.

There are essentially two categories of approaches proposed in the existing literature for handling deletion: *reset-recompute* [4, 31, 60, 61, 88], and *reachability-based* [45, 63, 88]. However, both approaches have performance limitations as they either incur redundant computation or require repeated whole-graph reachability analysis. In the rest of this section, we focus on these algorithms.

3.1 Reset-recompute Algorithm

Upon a deletion, a simple algorithm is to first reset the points-to sets of all variables that are "*relevant*" to the deleted statement and then recompute them following the same rationale as the on-the-fly algorithm. Here, "relevant" means "reachable" from the root variable of the change in the PAG.

Specifically, one can first remove from the PAG all edges related to the deleted statement and reset (*i.e.*, set to empty) the points-to sets of their destination nodes as well as all nodes that these destination nodes can reach (because the points-to sets of all those nodes may be affected). Then, for all the reset nodes, extract their associated points-to constraints and rerun the fixed-point computation following Algorithm 1.

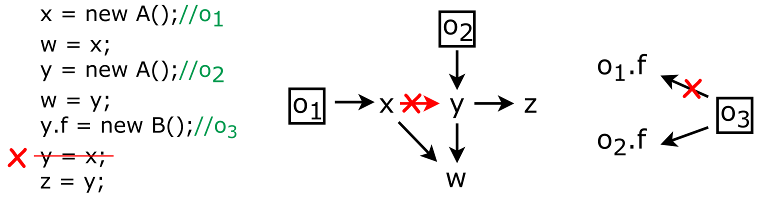

Fig. 3. An example of edge deletion in the PAG.

Consider an example in Figure 3, in which an edge $x \longrightarrow y$ is deleted from the PAG (e.g., due to the deletion of a statement $y = x$ in the program). The root variable of the change is y, since its points-to set may be changed immediately because of the edge deletion. The reset-recompute algorithm first resets $pts(y)$ as well as $pts(z)$ and $pts(w)$ to empty (because z and w are reachable from y). Then it extracts the points-to constraints $pts(y)= pts(y) \cup \{o_2\}$, $pts(z)= pts(z) \cup pts(y)$, $pts(w)= pts(w) \cup pts(y)$, and $pts(w)= pts(w) \cup pts(x)$, from the four edges connected to the three reset nodes, *i.e.*, $o_2 \longrightarrow y$, $y \longrightarrow z$, $y \longrightarrow w$ and $x \longrightarrow w$, and recomputes $pts(y)$, $pts(z)$ and $pts(w)$ until reaching a fixed point. The final values of the points-to sets are: $pts(x)=\{o_1\}$, $pts(y)=\{o_2\}$, $pts(z)=\{o_2\}$ and $pts(w)=\{o_1, o_2\}$. As o_1 has been removed from $pts(y)$ and y is the base variable of $y.f$, it also deletes the edge $o_3 \longrightarrow o_1.f$ and recomputes $pts(o_1.f) = \emptyset$.

The reset-recompute algorithm is inefficient because most computations on the points-to sets of the reset nodes may be redundant. For example, both before and after the statement deletion, $pts(w)$ remains the same. We call such recomputation on $pts(w)$ as *redundant computation* in the incremental analysis.

The idea of reset-recompute can be traced back to the DRed algorithm [22] for tabled evaluation and the work of Yur, Ryder and Landi [87] for pointer aliasing analysis. More recently, researchers have applied the reset-recompute algorithm to different dimensions of pointer analysis. We introduce two state-of-the-art approaches below.

IDE-/IFDS-based Algorithm. REVISER [4] proposes an interprocedural, flow- and context-sensitive data-flow analysis based on the IFDS/IDE (Interprocedural Distributive Environment Transformers) framework [57]. Similar to the *reset-recompute* algorithm, REVISER adopts a *clear-and-propagate* strategy to clear and recompute the relative analysis result when necessary.

The difference exists in the chosen frameworks. The IFDS framework formats the data-flow problem to a graph reachability problem. Furthermore, the IDE framework can solve distributive functions along the graph edges. REVISER can finish the incremental update within 50s for a single statement change in JUnit 4.10, For the code changes in a Soot [84] git commit, REVISER only requires 10s on average. We compare the performance of REVISER with IPA in Section 7.

Graph Pattern Matching. IncA [79] proposes a domain-specific language (DSL) to incrementally update program analysis result based on graph pattern matching. It uses *pattern functions* to define the relations between program entities in a program analysis (*e.g.*, the execution order of statements in control-flow analysis, the points-to relations of variables in points-to analysis).

IncA can be used to perform incremental points-to analysis by introducing a relation $PointsTo(x, y)$ to represent that variable x can point to variable y. For incremental statement changes, IncA initially updates the interprocedural control-flow graph (ICFG) incrementally. Then, according to the new control flow changes, it performs an incremental graph pattern matching (adopted from EMF-IncQuery [83]) to propagate changes to all dependent points-to relations and re-analyze the changed program entities. EMF-IncQuery is an incremental graph query engine to capture and execute live queries over EMF models (such as UML). It also supports incremental updates of model elements based on the Rete networks [19]. However, the specific algorithm for handling deletion is not very clear and may be determined by the considered patterns.

3.2 Reachability-based Algorithm

The basic idea of the reachability-based algorithm is to check the path reachability before removing an object from the points-to set of a variable. In other words, the points-to sets of those nodes which are potentially affected by the deletion are not reset, but are updated *lazily* only if they are not reachable from the nodes denoting the corresponding objects in the PAG. This algorithm does not incur any redundant computation on the points-to set, however, it requires repeated whole-graph reachability analysis, which is expensive for large PAGs.

Consider again the example in Figure 3. Upon the deletion of the edge $x \longrightarrow y$, the algorithm first checks if x is still reachable to y (*i.e.*, via another path without $x \longrightarrow y$). If yes, then the algorithm stops with no changes to any points-to set. Otherwise, it continues to check if any object in $pts(x)$ should be removed from $pts(y)$, by checking if the corresponding object node can reach y in the PAG. In this case, $pts(x)$ contains only o_1 which cannot reach y, hence o_1 is removed from $pts(y)$. Because $pts(y)$ is changed, the algorithm then continues to propagate the change by checking the nodes connected to y (*i.e.*, z and w). Because o_1 cannot reach z but can reach w (via the path $o_1 \longrightarrow x \longrightarrow w$), o_1 is removed from $pts(z)$ but $pts(w)$ remains unchanged. Finally, due to the change in $pts(y)$, it removes the edge $o_3 \longrightarrow o_1.f$, and checks if o_3 can still reach $o_1.f$. Since no incoming edge to $o_1.f$ exists, it removes o_3 from $pts(o_1.f)$.

The main scalability bottleneck of the reachability-based algorithm is that the worst case time complexity for checking path reachability is linear in the PAG size, which can be very large for real-world programs. For instance, in our experiments (Section 7) the PAG of the *h2* database contains over 300M edges, even with some JDK libraries excluded.

4 NEW INCREMENTAL ALGORITHMS

Our new algorithms are based on a fundamental *transitivity* property of Andersen's analysis. This enables us to prove two key properties of the PAG (with no cycles), which allow us to develop an efficient algorithm together with the incremental SCC optimization, without redundant computation or graph reachability analysis. We further prove an idempotency property of change propagation in pointer analysis, which allows parallelizing the incremental algorithm.

We first present the two key properties. We then present the basic incremental algorithm in Section 4.1 and the parallel incremental algorithm in Section 4.2.

According to Andersen's analysis rules in Table 1, we have the following correctness property:

Transitivity of PAG. For an object node o and a pointer node p in the PAG, $o \in pts(p)$ *iff* o can reach p. For two pointer nodes p and q, if p can reach q in the PAG, then $pts(p) \subseteq pts(q)$.

We first assume the PAG is *acyclic*, *i.e.*, all SCCs are collapsed into a single node and consider only *one* edge deletion. We will present our incremental SCC detection algorithm and describe our adaption of the on-the-fly Andersen's algorithm to handle multiple edge deletions in Section 4.1. Based on the transitivity property, we can prove the following lemma:

Lemma 1: Incoming neighbours property. Consider an acyclic PAG and a pointer node q of which an object $o \in pts(q)$. If q has an incoming neighbour r (*i.e.*, there exists an edge $r \longrightarrow q$) and $o \in pts(r)$, then there must exist a path from o to r without going through q.

Proof. See an illustration in Figure 4. First, because $o \in pts(r)$, due to transitivity, o can reach r. Second, because the PAG is acyclic, there cannot exist a path $o \longrightarrow \ldots \longrightarrow q \longrightarrow \ldots \longrightarrow r \longrightarrow q$ (which contains a cycle).□

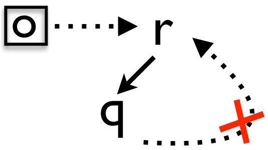

Fig. 4. Illustration of the incoming neighbours property.

Based on Lemma 1, we can prove the following theorem:

Theorem 1: *Suppose an edge $p \longrightarrow q$ is deleted from an acyclic PAG and all the other edges remain unchanged. For any object $o \in pts(q)$, if there exists an incoming neighbour r of q such that $o \in pts(r)$, then o remains in $pts(q)$. Otherwise if q does not have any incoming neighbour of which the points-to set contains o, then o should be removed from $pts(q)$.*

Proof. Due to Lemma 1, o can reach r without going through q. Hence, o can reach r without the edge $p \longrightarrow q$. Because $r \longrightarrow q$, o can hence reach q without the edge $p \longrightarrow q$. Therefore, o remains in $pts(q)$ after deleting $p \longrightarrow q$. Otherwise if no neighbour has a points-to set containing o, then o cannot reach q and hence should be removed from $pts(q)$.

With Theorem 1, to determine if a deleted edge introduces changes to the points-to information, we only need to check the incoming neighbours of the deleted edge's destination, which is much faster than traversing the whole PAG for checking the path reachability. Consider again the example in Figure 3. Upon deleting the edge $x \longrightarrow y$, we only need to check o_2, which is the only incoming neighbour of y. Because the points-to set of o_2 does not contain o_1, o_1 should be removed from $pts(y)$.

Once the points-to set of a node is changed, the change must be propagated to all its outgoing neighbours. Again, based on transitivity, we can prove the following lemma:

Lemma 2: Outgoing neighbours property. Consider an acyclic PAG and a pointer node q of which an object $o \in pts(q)$. If q has an outgoing neighbour w (*i.e.*, there exists an edge $q \longrightarrow w$) and w has an incoming neighbour r (different from q) such that $o \in pts(r)$. If r cannot reach q, then at least one of the following two conditions (or both of them) must hold in the PAG:

(1) There exists a path from o to w without going through q;

(2) There exists a path from q to r.

In other words, if every path from o to w must go through q, then there must exist a path from q to r; if there is no path from q to r, then there must exist a path from o to w without going through q.

Proof. See an illustration in Figure 5. There must exist such a path $o \rightarrow \ldots \rightarrow r \rightarrow w$ from o to w, because $o \in pts(r)$ and $r \rightarrow w$. The path may or may not contain q. However, if it contains q, then it must be $o \rightarrow \ldots \rightarrow q \rightarrow \ldots \rightarrow r \rightarrow w$, which means that q can reach r. It cannot be $o \rightarrow \ldots \rightarrow r \rightarrow \ldots \rightarrow q \rightarrow w$, because r cannot reach q.

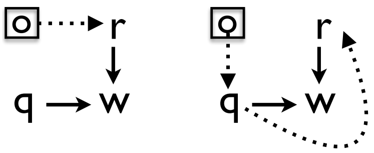

Fig. 5. Illustration of the outgoing neighbours property.

Based on Lemma 2, we can prove the following theorem:

Theorem 2: *Suppose an edge $p \rightarrow q$ was deleted from an acyclic PAG and it resulted in the removal of an object o from pts(q). To propagate this change, it is sufficient to check all the outgoing neighbours of q. For each outgoing neighbour w, if the points-to set of any of its incoming neighbours contains o, then the change propagation from this path to w can be skipped (the change may propagate to w again in the future from another path). Otherwise if none of the points-to sets of w's incoming neighbours contains o, o should be removed from pts(w) and the change should propagate further from w to all its outgoing neighbours.*

Proof. After the edge deletion, o was removed from $pts(q)$. Due to transitivity, o can no longer reach q in the remaining PAG. Consider an outgoing neighbour of q, w. If w has no incoming neighbour of which the points-to set contains o, then it means o cannot reach w and o should be removed from $pts(w)$. If w has an incoming neighbour r such that $o \in pts(r)$, we next prove that the change propagation from q to w can be skipped, while still ensuring the correctness of pointer analysis (*i.e.*, the transitivity of PAG).

Because o can reach r but cannot reach q, so r cannot reach q. Hence the condition of Lemma 2 is satisfied. Due to Lemma 2, there exists either (1) a path from o to w without going through q, (2) a path from q to r, or both (1) and (2). For (1), o should remain in $pts(w)$. This is satisfied vacuously following the change propagation rules in Theorem 2, because $pts(r)$ cannot be affected by the change propagation. For (2), following the outgoing neighbours of q, the change will propagate to r along a certain path and hence to w eventually. Therefore, to propagate change from a node, it is sufficient to check all the node's outgoing neighbours.

Theorems 1 and 2 together guarantee that upon deleting a statement, it suffices to check the local neighbours of the change impacted nodes in the PAG to determine the points-to set changes and to perform change propagation. This avoids redundant computations in recomputing the points-to sets and traversing the whole PAG.

Consider again the example in Figure 3. When o_1 is removed from $pts(y)$, we only need to check z and w, which are the outgoing neighbours of y. For z, because it does not contain any other incoming neighbour, o_1 is hence removed from $pts(z)$. However, for w, it has another incoming neighbour x (in addition to y) and $pts(x)$ contains o_1, so $pts(w)$ remains unchanged.

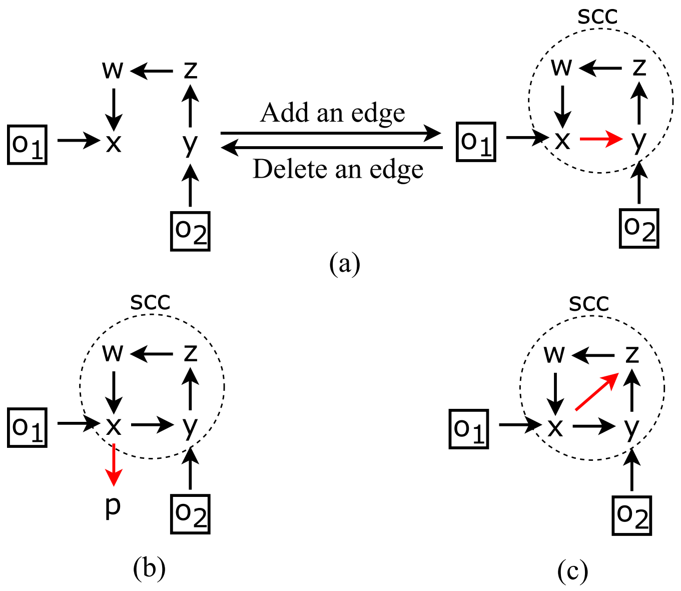

Fig. 6. Three SCC scenarios.

4.1 Basic Incremental Algorithm

In Theorems 1 and 2, we have made the assumption that the PAG is acyclic and we have considered only one edge deletion. The acyclic PAG can be satisfied by the SCC optimization, which is known in existing literature for whole program pointer analysis [23]. However, in the incremental setting, the SCCs must be dynamically updated. We first give a brief overview of our incremental SCC detection algorithm, which shares the same main idea with [7, 33]. Based on it, we then present our incremental algorithms for handling edge deletion and addition.

To support multiple edge deletions, we only need to slightly adapt the on-the-fly Andersen's algorithm (recall Algorithm 1). Specifically, we can change the on-the-fly algorithm such that within each iteration only a single edge deletion or insertion is applied. This does not affect the performance of the original algorithm because the same amount of computation is required to reach the fixed point.

4.1.1 Incremental SCC Detection. In incremental analysis, the main difference of the SCC optimization (from that in the on-the-fly Andersen's analysis) is that SCCs cannot only be augmented (by insertion), but also be broken (by deletion). An edge deletion may break a collapsed SCC into multiple smaller SCCs and/or individual nodes. In our algorithm, we maintain the collapsed SCCs and create a super node for each collapsed SCC in the PAG. For each deleted edge, we check the following conditions:

(1) The edge does not belong to any SCC: nothing to do with existing SCCs.
(2) The edge belongs to a certain SCC, but deleting the edge does not break the SCC. In this case, we keep the super node corresponding to the collapsed SCC in the PAG, and only remove the edge from the collapsed SCC. We use Tarjan's linear-time algorithm [81] to detect SCCs in the collapsed SCC after the edge deletion. If it returns the same SCC as the collapsed SCC, then it means the edge deletion does not break the existing SCC.

(3) The edge belongs to an SCC and removing it breaks the SCC. In this case, we first delete the super node corresponding to the collapsed SCC from the PAG, and restore all the nodes/edges in the broken SCC. Afterwards, we run Tarjan's linear-time algorithm inside the broken SCC and collapse any detected SCCs.

For each edge addition, we check the following conditions for the two nodes connected by the edge:

(1) If they belong to the same SCC, nothing to do with existing SCCs.
(2) If they do not belong to the same SCC, we use Tarjan's two-way search algorithm [6] for sparse graphs to detect new SCCs in the PAG incrementally. For each new SCC, we then collapse the SCC and create a new super node for it in the PAG. Any existing SCCs contained in the new SCCs are removed.

Figure 6 illustrates the incremental SCC detection with examples. Figure 6(a) shows that adding the edge $x \longrightarrow y$ creates a new SCC and deleting the edge breaks the SCC. Figure 6(b) shows that the edge $x \longrightarrow p$ does not belong to any SCC, so adding/deleting the edge does not create new SCCs or affect existing SCCs. Figure 6(c) shows that the edge $x \longrightarrow z$ belongs to an SCC, but adding/deleting it does not augment or break the SCC.

4.1.2 Incremental Edge Deletion. Algorithm 2 shows our incremental algorithm for handling edge deletion. We maintain a PAG and a worklist, which is initialized to the input deleted edge. In each iteration, one edge from the worklist is processed, which involves two steps. First, we remove the edge from the PAG and handle the SCCs according to the incremental SCC detection algorithm described in Section 4.1.1. We ensure that after deleting the edge the PAG is acyclic and all SCCs are collapsed into a single node.

After that, we run the procedure *PropagateDeleteChange* to propagate the points-to set changes caused by the edge deletion. This procedure takes two inputs: a set Δ of potential points-to set changes, and a node y that these changes are propagating to. For an edge $x \longrightarrow y$, Δ is initialized to $pts(x)$, because after deleting the edge all objects in $pts(x)$ may be removed from $pts(y)$. Then, we check the incoming neighbours of y; if any change in Δ is contained in the points-to set of a neighbour, the change should be skipped, *i.e.*, not applied to $pts(y)$. Hence, we remove from Δ all the objects that overlap with the points-to sets of y's incoming neighbours. For the remaining objects in Δ, we then remove them from $pts(y)$ and propagate them further to all of y's outgoing neighbours.

To handle those dynamic edges that can be deleted during the change propagation, we run the procedure *CheckNewDeletedEdges* once any change is applied to a node, *i.e.*, any object is removed from or added to its points-to set. This procedure takes a points-to set change and a target node as input, and returns a list of deleted PAG edges to the worklist. Note that the complex statements (*i.e.*, load, store and call) can introduce new edges. Now, we are processing PAG edge deletion. In *CheckNewDeletedEdges*, for each object $o \in \Delta$, that is removed from $pts(y)$ and for each node $o.f$ in the PAG that is generated from $y.f$, we remove all edges from/to $o.f$ (because the node $o.f$ should be removed). For a deleted method call $a = b.m(c)$, we simply remove the edges $c \longrightarrow p$ and $r \longrightarrow a$ (p is the formal parameter and r the return variable of m), which are introduced to the PAG when the method call is added. Note that the nodes/edges of the method body remain unchanged. This not only addresses multiple calls to a method in the same context, but also improves performance when the method call is added back later.

4.1.3 Incremental Edge Addition. Algorithm 3 shows our incremental algorithm for handling edge insertion, which follows the on-the-fly algorithm in Algorithm 1. Compared with our incremental deletion algorithm, it has three main differences. First, instead of deleting edges from the PAG, it always adds edges. Second, it does not need to check incoming neighbours. To propagate a change to a node, it simply checks if the node's points-to set contains the change or not. If yes the change is

ALGORITHM 2: DeleteEdge(e)

```
Input  : e - a deleted edge
         pag - the PAG
1  $WL \leftarrow e$ // initialize worklist to $e$
2  while $WL \neq \emptyset$ do
3      $e \leftarrow$ RemoveOneEdgeFrom($WL$)
4      $pag \leftarrow pag \setminus \{e\}$
5      DetectSCC($e$)
       // let $e$ be $x \rightarrow y$
6      PropagateDeleteChange($pts(x)$, $y$)
7  end

8  PropagateDeleteChange($\Delta$, $y$):
   Input  : $\Delta$ - a set of points-to set changes
            $y$ - a node that $\Delta$ propagates to
9  foreach $z \rightarrow y$ do //$z$ is an incoming neighbour of $y$
       // Objects in $\Delta$ but not in $pts(z)$
10     $\Delta = \Delta \setminus (\Delta \cap pts(z))$
11     if $\Delta = \emptyset$ then
12         return
13     end
14 end
   // remove $\Delta$ from $pts(y)$
15 $pts(y) \leftarrow (pts(y) \setminus \Delta)$
16 foreach $y \rightarrow w$ do //$w$ is an outgoing neighbour of $y$
17     PropagateDeleteChange($\Delta$, $w$)
18 end
19 $WL \leftarrow$ CheckNewDeletedEdges($\Delta$, $y$)

20 CheckNewDeletedEdges($\Delta$, $y$):
21 foreach $o \in \Delta$ do
       // process complex statements related to $y.f$
22     foreach node $o.f$ generated from $y.f$ do
           // add to $WL$ all edges from/to $o.f$
23         $WL \leftarrow e$ // let $e$ be $o.f \rightarrow *$ or $* \rightarrow o.f$
24     end
25 end
```

skipped, otherwise the change is applied. Third, once the points-to set of a node is changed, it checks if the node corresponds to a base variable in any complex statement and adds new edges to the PAG correspondingly. The *CheckNewAddedEdges* procedure takes a points-to set change Δ and a target node y as input, and returns a list of added PAG edges to the worklist. For each object $o \in \Delta$, for a *load* statement $x = y.f$ we add a new edge $o.f \longrightarrow x$; for a *store* statement $y.f = x$, we add a new edge $x \longrightarrow o.f$; and for a *call* statement $y.m()$, we check the method $T.m()$ (where T is the type of the object o) and add the corresponding method call edges, and also analyze the method body if $T.m()$ is new.

ALGORITHM 3: AddEdge(e)

```
Input  : e - an inserted edge
         pag - the PAG
1  $WL \leftarrow e$ // initialize worklist to $e$
2  while $WL \neq \emptyset$ do
3      $e \leftarrow$ RemoveOneEdgeFrom($WL$)
4      $pag \leftarrow pag \cup \{e\}$
5      DetectSCC($e$)
       // let $e$ be $x \rightarrow y$
6      PropagateAddChange($pts(x)$, $y$)
7  end

8  PropagateAddChange($\Delta$, $y$):
   Input  : $\Delta$ - a set of changes
            $y$ - a node that $\Delta$ propagates to
   // Objects in $\Delta$ but not in $pts(y)$
9  $\Delta = \Delta \setminus (\Delta \cap pts(y))$
10 if $\Delta \neq \emptyset$ then
       // add $\Delta$ to $pts(y)$
11     $pts(y) \leftarrow (pts(y) \cup \Delta)$
12     foreach $y \rightarrow w$ do //$w$ is an outgoing neighbour of $y$
13         PropagateAddChange($\Delta$, $w$)
14     end
15     $WL \leftarrow$ CheckNewAddedEdges($\Delta$, $q$)
16 end

17 CheckNewAddedEdges($\Delta$, $y$):
18 foreach $o \in \Delta$ do
       // process complex statements related to $y.f$
19     foreach Load $x = y.f$ do
           // add a new edge to $WL$
20         $WL \leftarrow e$ // let $e$ be $o.f \rightarrow x$
21     end
22     foreach Store $y.f = x$ do
           // add a new edge to $WL$
23         $WL \leftarrow e$ // let $e$ be $x \rightarrow o.f$
24     end
25     foreach Call $y.m()$ do
           // add new edges in $o.m$ to $WL$
26         $WL \leftarrow$ AnalyzeNewMethod($o.m$)
27     end
28 end
```

4.2 Parallel Incremental Algorithm

Our parallel incremental algorithms are based on a strong *change idempotency* property of our basic incremental algorithms described in Section 4.1.

Lemma 3: Change idempotency property: For an edge insertion or deletion, the update to each points-to set is an idempotent operator. In other words, if the change propagates to a node more than once from different paths, the effect of the change (*i.e*, the modification applied to the corresponding points-to set) must be the same.

Proof. Suppose two changes Δ_1 and Δ_2 are propagated to the same node q along two different paths: $p \to \ldots \to r_1 \to q$ ($path_1$) and $p \to \ldots \to r_2 \to q$ ($path_2$), respectively, where p is the root change node (the insertion or deletion of an edge ending at p) and r_1 and r_2 are the two incoming neighbours of q. And suppose that there exits an object o such that $o \in \Delta_1$ and $o \notin \Delta_2$.

For deletion, we can prove that there must exist a node w on $path_2$ such that o is reachable to w without going through p (otherwise, the deletion of o would have propagated to r_2, which contradicts with $o \notin \Delta_2$). Due to transitivity, we have $o \in pts(r_2)$. Because r_2 is an incoming neighbour of p, o will not be removed from $pts(p)$. In other words, any object $o \notin \Delta_1 \cap \Delta_2$ will be preserved in $pts(p)$. Therefore, the changes applied to $pts(q)$ are always the same.

For addition, we can prove that o must be contained in $pts(q)$. The reason is that both Δ_1 and Δ_2 must be originated from the same root change Δ and o must be in Δ. If o is not in Δ_2, then there must exist a node w on $path_2$ such that $o \in pts(w)$, and again due to transitivity, $o \in pts(q)$. In other words, any object $o \notin \Delta_1 \cap \Delta_2$ should be already included in $pts(p)$. Therefore, the changes applied to $pts(q)$ are always the same.

Based on Lemma 3, in each iteration of our incremental algorithm, we can parallelize the change propagation along different paths with no conflicts (if atomic updates are used). More specifically, we can propagate the points-to set change of a node along all its outgoing edges in parallel without worrying about the order of propagation. Moreover, because concurrent modifications to the same points-to set are always consistent, we do not even need synchronization among them.

ALGORITHM 4: ParallelPropagateDeleteChange(Δ, y)

```
Input  : $\Delta$ - a set of changes
         $y$ - a node that $\Delta$ propagates to
1  foreach $z \to y$ do
2      $\Delta = \Delta \setminus (\Delta \cap pts(z))$
3      if $\Delta = \emptyset$ then
4          return
5      end
6  end
7  $pts(y) \leftarrow (pts(y) \setminus \Delta)$
   // all outgoing edges in parallel
8  Parallel foreach $y \to w$ do
9      ParallelPropagateDeleteChange($\Delta$, $w$)
10 end
11 sync {$WL$} $\leftarrow$ CheckNewEdges($\Delta$, $y$)
```

Algorithms 4 and 5 show our parallel incremental algorithms for deletion and insertion, respectively. We propagate the points-to set change of a node along all its outgoing edges in parallel (see line 8 in Algorithm 4 and line 4 in Algorithm 5). Our algorithm guarantees that a change can only propagate through a node at most once, even though there might be multiple parallel propagation paths reaching the same node. Figure 7 shows an example. Initially, $pts(y) = pts(q) = \{o_1\}$ and $pts(p) = pts(z) = pts(w) = \{o_1, o_2\}$. After deleting the edge $x \to y$, $pts(y)$ is updated to $\{o_2\}$, and the change $\{o_1\}$ is then propagated from y to all the other nodes that y can reach (*i.e.*, p, q, z and w).

ALGORITHM 5: ParallelPropagateAddChange(Δ, y)

```
Input  : $\Delta$ - a set of changes
         $y$ - a node that $\Delta$ propagates to
1 $\Delta = \Delta \setminus (\Delta \cap pts(y))$
2 if $\Delta \neq \emptyset$ then
3     $pts(y) \leftarrow (pts(y) \cup \Delta)$
      // all outgoing edges in parallel
4     Parallel foreach $y \rightarrow w$ do
5         ParallelPropagateAddChange($\Delta$, $w$)
6     end
7     sync $\{WL\} \leftarrow$ CheckNewEdges($\Delta$, $y$)
8 end
```

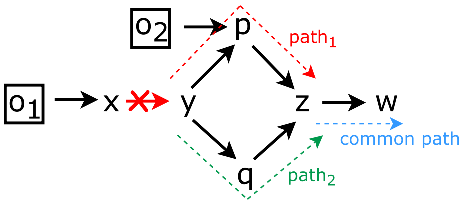

Fig. 7. An example of parallel change propagation.

Based on Algorithm 4, the change propagates along the two paths (*i.e.*, $path_1$ and $path_2$) in parallel, and reaches a common node z. There are three possibilities to consider in this process:

- The propagation along $y \rightarrow p$ completes faster than that along $y \rightarrow q$. At this time, o_1 has been removed from $pts(p)$, and we are still checking the incoming neighbours of q, where $pts(q) = \{o_1\}$. Then, the propagation from $path_1$ reaches z. Since q is an incoming neighbour of z and $o_1 \in pts(q)$, $pts(z)$ will not be changed and hence the propagation from $path_1$ terminates at z. Later, when the propagation from $path_2$ reaches z, because $pts(q)$ and $pts(q)$ do not contain o_1 anymore, o_1 is finally removed from $pts(z)$ and the change propagates further to w from $path_2$.
- The propagation along $y \rightarrow q$ completes faster than that along $y \rightarrow p$. Opposite to the first case, the change propagation from $path_2$ terminates at z, and the change propagation from $path_1$ continues through z.
- The propagation along both paths reaches p and q at the same time. At this time, $pts(p) = \{o_2\}$ and $pts(q) = \emptyset$. Both changes from p and q propagate to z. No matter which propagation reaches z first, o_1 will be removed from $pts(z)$. When the later propagation comes, no change to $pts(z)$ will be performed, since o_1 has already been removed from $pts(z)$. If both of them reach z at the same time and both attempt to remove o_1 concurrently, to minimize the amount of the points-to set computation, we can synchronize the updates of $pts(z)$, such that only one of them can succeed and can continue the change propagation.

In summary, $pts(z)$ needs to be updated once, regardless of the parallel propagation schedule.

In addition to the points-to set, the worklist (line 11 in Algorithm 4 and line 7 in Algorithm 5) is synchronized, because different parallel tasks may concurrently add different new edges to the worklist.

4.2.1 Synchronization-free Implementation. In practice, we would like to avoid synchronizations as much as possible, since synchronizations on parallel processors are expensive. We propose a synchronization-free implementation of the points-to set data structure. The limitation is that concurrent updates to the same points-to set may all succeed, which may lead to redundant propagations. Nevertheless, since the chance is very small for a change to propagate from multiple paths to the same node at the same time, this optimization works well in practice.

Our implementation maintains an entry for each object `o` and supports three operations: `contains(o)`, `add(o)` and `remove(o)`. In both `add(o)` and `remove(o)`, a flag is returned to indicate whether the change was successful (if not, another thread has already done this). This flag can then be used to prevent unnecessary further propagation from the thread that came second. We next show why no synchronization is required for these points-to set operations.

Suppose two threads T1 and T2 concurrently execute Algorithm 4 or Algorithm 5, there are only four possible conflicting scenarios and each scenario always produces a consistent result regardless of synchronization or any atomicity requirement of the three operations:

(1) In Algorithm 4, T1:`contains(o)` at line 2 on $pts(r)$ and T2:`remove(o)` at line 7 on $pts(z)$. Consider the operation `contains(o)` by T1. With or without synchronization, it always returns either true or false. If false, then `o` will be removed from $pts(y)$ at line 7 by T1. If true, `o` will not be removed from $pts(y)$ by T1; however, `o` is removed from $pts(z)$ by T2 and because y is an outgoing neighbour of z the change will propagate to y. Finally, `o` will be removed from $pts(y)$ by T2 or by another thread.
(2) In Algorithm 4, both T1:`remove(o)` and T2:`remove(o)` at line 7 on $pts(y)$. The entry for o in $pts(y)$ will be set to 0 (meaning o is not included) by both T1 and T2, *i.e.*, `o` will be removed from $pts(y)$.
(3) In Algorithm 5, T1:`contains(o)` at line 1 on $pts(y)$ and T2:`add(o)` at line 3 on $pts(y)$. The operation `contains(o)` by T1 may return either true or false. If false, then `o` will be added to $pts(y)$ at line 7 by T1. If true, `o` has already been added to $pts(y)$ by T2.
(4) In Algorithm 5, both T1:`add(o)` and T2:`add(o)` at line 3 on $pts(y)$. The entry for o in $pts(y)$ will be set to 1 (meaning o is included) by both T1 and T2, *i.e.*, `o` will be added to $pts(y)$.

5 END-TO-END INCREMENTAL POINTER ANALYSIS

In this section, we present IPA, an end-to-end incremental pointer analysis for real-world Java programs based on our new incremental algorithms described in Section 4. The presented pointer analysis is context-, path- and flow-insensitive.

Consider a programming environment where the developer has performed an initial commit of her project, we compile the project, translate the Java bytecode to an SSA-based IR [5] and use the IR to construct a PAG. Then, the developer commits a collection of program changes $\Delta_{P_{src}}$. We recompile the project with $\Delta_{P_{src}}$ to obtain the updated IR, compare with the old IR to obtain the IR changes $\Delta_{P_{IR}}$, which contains multiple new IR statement insertions and/or old IR statement deletions or modifications[1]. $\Delta_{P_{IR}}$ can be divided into two disjoint sets: D - a set of old IR statement deletions and I - a set of new IR statement insertions. In our implementation, we use the Java Source Front End provided in WALA to translate the changed source file to Eclipse AST and then to IR (which is very fast, typically within 10ms for a changed method). If the source file is uncompilable, e.g., because of syntax error or missing dependencies, IPA will not run.

[1]A modification of existing IR statements can be treated as deletion of the old IR statements and insertion of the new IR statements, and a large code chunk can be treated as a collection of small changes.

ALGORITHM 6: Incremental Pointer Analysis for Java

```
Input  : $\Delta_{P_{IR}}$ - a set of IR changes.
         Deletions: D: -{d1,d2,...};
         insertions: I: +{i1,i2,...}.
  // for each deleted IR statement
1 foreach $s \in D$ do
      // extract edge(s) according to Table 2
2     $e \leftarrow$ ExtractEdge($s$)
      // call Algorithm 2 for each deleted edge
3     DeleteEdge($e$)
4 end
  // for each inserted IR statement
5 foreach $s \in I$ do
      // extract edge(s) according to Table 2
6     $e \leftarrow$ ExtractEdge($s$)
      // call Algorithm 3 for each added edge
7     AddEdge($e$)
8 end
```

Table 2. Java statements and their corresponding PAG edges.

Statement	PAG Edges
❶ `x = new T()`	$o \longrightarrow x$
❷ `x = y`	$y \longrightarrow x$
❸ `x = y.f`	$y.f \longrightarrow x$ & $\forall o \in pts(y)$: $o.f \longrightarrow x$
❹ `x.f = y`	$y \longrightarrow x.f$ & $\forall o \in pts(x)$: $y \longrightarrow o.f$
❺ `x = y[i]`	$y.* \longrightarrow x$ & $\forall o \in pts(y)$: $o.* \longrightarrow x$
❻ `x[i] = y`	$y \longrightarrow x.*$ & $\forall o \in pts(x)$: $y \longrightarrow o.*$
❼ `a = b.m(c)**`	$c \longrightarrow p$ & $r \longrightarrow a$

**(Suppose a and c are both reference variables and p is the formal parameter and r the return variable of m).

Algorithm 6 shows our end-to-end algorithm. For each IR statement, we first extract the corresponding edges in the PAG according to Table 2. In Java, there are seven types of statements that must be analyzed for pointer analysis. Each statement corresponds to one or more edges in the PAG:

❶ (**allocate**): an edge from an object node o to a pointer node x. o is identified by its allocation site and has a type T.
❷ (**simple assignment**): an edge from a pointer node y to x.
❸ (**field load**): an edge from a pointer node $y.f$ to x, and for each object o in $pts(y)$, an edge from $o.f$ to x.
❹ (**field store**): an edge from a pointer node y to $x.f$, and for each object o in $pts(x)$, an edge from y to $o.f$.
❺ (**array load**): an edge from a pointer node $y.*$ to x, and for each object o in $pts(y)$, an edge from $o.*$ to x. For array load ❺ and store ❻, since we do not perform array index analysis, different array elements are not distinguished but represented by a special constant index "$*$".
❻ (**array store**): an edge from a pointer node y to $x.*$, and for each object o in $pts(x)$, add an edge from y to $o.*$.

❼ (**method call**): an edge from a pointer node c to the method m's formal parameter node p, and an edge from m's return node r to a.

For statements ❶-❻, their treatments are the same as that in Andersen's analysis (Table 1). For method call ❼, for each $o \in pts(b)$, if $T.m$ (where T is the type of o) corresponds to a new method, all statements in the method body are also analyzed. For most other types of statements such as loops and branches, we can simply ignore them because our analysis is path- and flow-insensitive. However, analyzing them may improve the precision of pointer analysis.

For each identified edge, we then call Algorithm 2 if it is deleted and Algorithm 3 if added, to compute the new points-to information and update the PAG. To parallelize our algorithm, in each iteration of Algorithms 2 and 3, we call Algorithms 4 and 5 to propagate the points-to set changes in parallel. On a multicore machine, we can maintain a thread pool to perform the parallel tasks.

Dynamic Language Features. As IPA analyzes code statically, it is difficult to handle dynamic language features such as dynamic class loading and reflection in Java. This is a known challenge in static analysis, and previous research has proposed a number of techniques (notably [9, 40, 43, 69]) to address it. In our implementation, we rely on WALA's existing support to deal with dynamic features. For example, WALA has a significant support for reflective constructs by using the concrete type of the receiver class in a context-sensitive way [29].

6 DISCUSSIONS

6.1 Adapting to Context-sensitive, Flow-sensitive and Other Problems

Heretofore, our incremental and parallel algorithm is field-sensitive, but context-insensitive and flow-insensitive. Next, we discuss the potential to extend our algorithm to other dimensions in pointer analysis and other research problems.

Context-sensitivity. We note that our incremental algorithms can also apply to context-sensitive pointer analysis because the handling of edge insertions and deletions is orthogonal to the representation of context. In general, there are two types of techniques in context-sensitive pointer analysis: k-CFA [38, 64, 72] and CFL-reachability [45, 63, 71, 74].

In a k-CFA pointer analysis, pointer variables uses k call strings from the call graph to distinguish contexts. When there is an incremental code change, we can follow Algorithms 2 and 3 to perform the update for each edge with an additional criterion: we need to consider all the k call strings of the changed node.

CFL-reachability pointer analysis also represents a program by a PAG, which includes load/store edges to indicate field accesses and entry/exit edges to indicate the calling context. We need to discover a feasible path between an object node and a variable node with matching edge labels in the PAG in order to compute its points-to set. Although the CFL-reachability paths are not transitive closures, the transitive closure property may still be maintained by using a trace-based incremental CFL-reachability algorithm [45]. The idea is that the computed feasible paths for answering a points-to query can be cached as traces. In the cached traces, the transitive closure property still holds. When a change occurs, we can check the path feasibility of the traces that contain the changed node, and as long as the path is still feasible, we can apply Algorithms 2 and 3 along the cached traces to incrementally update the points-to results.

Besides, CFL-reachability pointer analysis always computes points-to sets in a demand-driven way. For this case, we only need to update an edge with its labels in the PAG. Then, discover all reachable paths containing the edge, and propagate the change along these paths if any answer for a variable on the paths has been cached.

Flow-sensitivity. Flow-sensitive pointer analysis is often a lot more expensive and complicated than its flow-insensitive version. To determine a points-to set at a program point, a partial SSA representation and control flow graph can be combined to infer points-to and def-use relations and to avoid unnecessary propagation, which can also be represented by graphs [13, 24, 25]. Sometimes, the analysis can be cast to a graph reachability problem [39].

When incremental changes are introduced to a program, we need to consider: (1) the introduced control flow changes and its corresponding def-use changes, (2) a strong update or weak update on the affected points-to sets, and (3) interprocedural data flow changes. Since the points-to relation and control flow are coupled in the graph when considering flow-sensitivity, a simple check and propagation on local neighbours in the graph cannot guarantee a precise result.

A possible solution is to decouple the graph that encodes points-to and control flow as proposed in IncA [79], in which points-to graph and control-flow graph are maintained separately. In this case, a two-step update can be performed: first, update the def-use information according to the control flow changes; second, update the points-to relation based on the new def-use and program changes. Hence, our incremental algorithm can be modified to perform on a single graph in each step. However, we note that IncA may loss precision because the constructions of a precise control-flow graph and a precise points-to analysis are typically recursively dependent on each other.

Other Research Problems. Our incremental and parallel analysis can be adapted to work on other research problems that require analyzing and updating information on dynamic graphs, as long as the analyzed graph $G = (V, E)$ satisfies the following properties:

- G is a directed graph, and SCC collapsing can be applied on G to obtain an acyclic graph, with or without precision loss on the property of interest;
- Each node in V represents a set of elements, and each edge in E represents a constraint which propagates the elements;
- The constraint satisfies the local neighbour properties presented in Section 4;
- The direction on an edge indicates the propagation direction;
- The propagation of elements on G can terminate.

For example, dynamic algorithms for directed graph reachability have been used in motion planning to search for the next move in the configuration space [53] and to identify collision-free paths [46]. Our algorithms may be applied there to improve efficiency.

6.2 Scheduling of Changed Statements

Since multiple statement insertions and deletions may happen at one time and sometimes an insertion can invalidate a deletion effect, the order of statement processing may affect the performance of an end-to-end incremental pointer analysis. Some optimizations (*e.g.*, scheduling of updates [62]) can be performed to reduce such redundant work.

Our pointer analysis is built on SSA-based IR, where each variable and its corresponding pointer node in a method are named by a unique value number (*e.g.*, $v1$, $v2$, $v3$) rather than by its variable name (*e.g.*, `a`, `x`, `y`). After several statement insertions and deletions, new value numbers have been assigned to the variables in the updated code. Hence, it is difficult to identify the correspondence between an added new statement and a deleted old statement. Even though they have the same value number, the number may represent different variables in the method. Rather than performing a complex procedure to identify the correlations between each added and deleted statements, we use a straightforward strategy as described in Algorithm 6, in which we handle deletion followed by addition. Nevertheless, we note that this is only an engineering choice. With an optimization to find the correlations, we expect that the efficiency of our algorithm can be further improved.

7 EVALUATION

We implemented IPA in WALA [85], a popular static program analysis framework that supports a variety of pointer analyses. We modified the *ZeroOneContainerCFA* pointer analysis, which uses a single InstanceKey for every allocation site. To compare with the existing classic algorithms, we also implemented the reset-recompute and the reachability-based algorithms in the same framework. These two algorithms are based on our prior work, ECHO [88], an IDE-based incremental bug detection tool for detecting data races. To compare with the state-of-the-art tools, we also performed an evaluation of REVISER[2] [4]. We evaluated IPA and REVISER on a collection of 14 real-world large Java applications from DaCapo-9.12, as shown in Table 3. However, we note that this is not an apples-to-apples comparison because IPA implements an incremental pointer analysis, while REVISER implements different incremental client analyses over a non-incremental pointer and call-graph analysis. Moreover, REVISER is context- and flow-sensitive, while IPA is not. In this section, we report the results of our experiments.

Evaluation Methodology. For each benchmark, we run three sets of experiments. (1) We first run the whole program exhaustive pointer analysis (*i.e.*, the default ZeroOneContainerCFA) to construct the PAG. Then, we initialize IPA with the PAG computed for the whole program in the first step and continue to conduct two experiments with incremental code changes. (2) For each statement in each method in the program, we delete the statement and run IPA. (3) For the deleted statement in the previous step, we add it back and re-run IPA.

We run IPA with two configurations: with a single thread (*IPA-1*) to evaluate our basic incremental algorithms only, and with a pool of 48 threads (*IPA-48*) to evaluate our parallel incremental algorithms. We measure the time taken by each component in each step and compare the performance between the exhaustive analysis and IPA. We repeat the same experiments for the reset-recompute and the reachability-based algorithms to compare their performance with IPA. For the evaluation of REVISER, we initially compute its data-flow analysis for the whole program and obtain its points-to information using the default points-to analysis. Then, we repeat the same experimental procedure. The whole program optimizations has been turned on during the entire process. For this evaluation, the incremental performance we report includes both the time to re-compute the call graph, and the time to incrementally update the ICFG and PAG.

In addition, we replace the default incremental pointer analysis algorithm in ECHO (which is a hybrid implementation of reset-recompute and reachability-based) with IPA and compare the time taken by the tool for detecting data races.

Since it is computational expensive to run the incremental analysis experiments (2) and (3) for all statements in the benchmarks (which would take several months to years), in each configuration we limit the total time for each benchmark to two hours. As a result, the performance numbers correspond to those statements that are analyzed within two hours.

All experiments were performed on an HPC server with Dual 12-core Intel Xeon CPU E5-2695 v2 2.40GHz (2 threads per core) processors and 755GB of RAM. The JDK version was 1.8.

We set the maximum heap to 755GB to run all experiments. Since we want to emphasize the performance improvement of our incremental and parallel pointer analysis over the exhaustive analysis and other existing incremental sequential techniques, we do not provide an evaluation on memory usage among *IPA-1*, *IPA-48* and other techniques.

Benchmarks. The metrics of the benchmarks and their PAGs are reported in Table 3. Columns 2-6 report the numbers of classes, methods, pointer nodes, object nodes and edges in the PAGs, respectively. More than half of the benchmarks contain over 1M pointer nodes and over 200M edges

[2]REVISER is available at https://github.com/secure-software-engineering/reviser

Table 3. Benchmarks and the PAG metrics.

Benchmark	#Class	#Method	#Pointer	#Object	#Edge
avrora	23K	238K	2M	33K	229M
batik	23K	60K	1.2M	31K	272M
eclipse	21K	36K	365K	7K	44M
fop	19K	68K	2M	42K	295M
h2	20K	69K	2M	32K	301M
jython	26K	79K	2M	53K	325M
luindex	20K	71K	1.8M	29K	299M
lusearch	20K	63K	1M	18K	185M
pmd	22K	42K	983K	25K	101M
sunflow	22K	73K	1.5M	32K	218M
tomcat	16K	36K	886K	23K	94M
tradebeans	14K	39K	674K	19K	99M
tradesoap	14K	38K	653K	20K	97M
xalan	21K	33K	576K	15K	138M

in the PAG. The default ZeroOneContainerCFA pointer analysis creates an object node for every allocation site and has unlimited object-sensitivity for collection objects. For all benchmarks certain JDK libraries such as *java.awt.** and *java.nio.** are excluded[3] to ensure that the exhaustive pointer analysis analysis can finish within 6 hours.

Empirical correctness. Besides the performance experiments, we also empirically validated the correctness of our implementations of the incremental algorithms. We set up the tooling to also compare the points-to results of different incremental and the exhaustive algorithms whenever possible. More specifically, after each round of deletion and insertion experiments for a statement, we check if the points-to result remains the same as before. In addition, we cross validate the correctness of the incremental algorithms by checking the updated points-to results after each deletion. Because re-running the exhaustive analysis after every deletion is too time-consuming, we have also checked for several outstanding deletions only (*i.e.*, those special cases that take ≥1s to handle and cause changes in the points-to result) and confimred that all the compared incremental algorithms compute the same points-to result as the exhaustive analysis.

7.1 Performance of Incremental Deletion

Table 4 compares the performance between the exhaustive pointer analysis (*Full*) and the incremental algorithms for deletion. Overall, IPA achieves dramatic speedups over the other algorithms. On average, IPA is two to five orders of magnitude faster than the exhaustive algorithm and two orders of magnitude faster than the other two existing incremental algorithms (*i.e.*, reset-recompute and reachability-based) and REVISER. For most benchmarks, the exhaustive analysis takes several hours to compute (2.4h on average). For a deletion change, the *reset-recompute* algorithm takes 26s on average, the *reachability-based* algorithm takes 39s, whereas *IPA-1* takes only 73ms to analyze, which is at least 300X faster than the other algorithms. REVISER shows an impressive performance (*e.g.*, 153s for all cases on average and 268s for the worst cases), especially for the worst-case scenarios. However, it still two orders of magnitude slower than *IPA-1*. REVISER did not perform a successful run on *jython* (indicated by "-" in Table 4 and 5), due to an unrecognized Java type in the Jimple IR construction. Compared to *IPA-1*, *IPA-48* further improves performance by an order

[3]This can be done by configuring the EclipseDefaultExclusions.txt file in WALA. Analyzing the entire program including all those libraries in WALA typically takes a long time (>6h without finishing) or runs out of memory.

Table 4. Performance of the incremental algorithms for handling deletion.

Benchmark	Full	Reset-Recmpt		Reachability		REVISER		IPA-1		IPA-48	
		Avg	Worst	Avg	Worst	Avg	Worst	Avg	Worst	Avg	Worst
avrora	4.8h	52s	30+min	76s	30+min	135s	230s	89ms	228s	27ms	10.5s
batik	4.1h	48s	22min	79s	30+min	351s	634s	95ms	51s	42ms	6.1s
eclipse	1h	14s	12min	20s	15min	120s	201s	65ms	21s	23ms	2.6s
fop	3.3h	31s	16min	38s	21min	110s	202s	110ms	172s	29ms	5.9s
h2	3.9h	37s	25min	82s	30+min	133s	236s	78ms	120s	24ms	9.4s
jython	3.2h	43s	17min	67s	30+min	-	-	96ms	480s	18ms	22s
luindex	2.9h	22s	10min	31s	12min	127s	234s	143ms	162s	31ms	7s
lusearch	2.5h	17s	7min	11s	8min	117s	218s	15ms	44s	9ms	2.8s
pmd	0.65h	14s	30+min	14s	30+min	240s	346s	67ms	27s	13ms	1.2s
sunflow	3.5h	47s	11min	61s	18min	221s	322s	66ms	90s	36ms	8s
tomcat	0.6h	9.8s	30+min	12s	30+min	115s	215s	64ms	19s	28ms	1.8s
tradebeans	0.75h	3.5s	7min	3s	9min	109s	223s	24ms	14s	11ms	0.8s
tradesoap	0.8h	4s	10min	5s	11min	112s	217s	31ms	18s	15ms	1s
xalan	0.47h	38s	30+min	14s	8min	104s	205s	12ms	13s	5ms	1.7s
Average	2.4h	26s	17+min	39s	22+min	153s	268s	73ms	66s	24ms	5.5s

of magnitude. *IPA-48* takes only 24ms on average, more than three orders of magnitude faster than existing incremental algorithms.

The speedup is also significant for the worst case scenarios, where analyzing a certain deletion change takes the longest time among all changes in each benchmark. In the worst case, *reset-recompute* takes more than 17mins, *reachability* takes more than 22mins and REVISER needs 268s on average, while *IPA-1* takes 66s and *IPA-48* takes 5.5s only on average per deletion change, achieving more than 200X speedup over existing algorithms.

The worst case scenarios are often hundreds to thousands of times slower than the average cases in the two incremental algorithms from ECHO. For example, for *Avrora*, the exhaustive analysis takes 4.8h, and the four incremental algorithms (reset-recompute, reachability-based, *IPA-1* and *IPA-48*) take 52s, 76s, 89ms and 27ms respectively per deletion on average. However, for the slowest case, the four incremental algorithms take 30+mins, 30+mins, 228s and 10.5s respectively. The reason is that there is often a small number of complex array load and store statements in each benchmark, which involve a long chain of dependencies and a large points-to set. These statements produce a large number of edge changes upon their deletions and incur long change propagation paths. However, such special cases are very rare. As we will discuss in Section 7.3, for more than 99.9% of the statements, IPA takes under 1s.

7.2 Performance of Incremental Addition

Table 5 compares the performance between the exhaustive pointer analysis and the incremental algorithms for insertion. We note that the existing incremental approaches for handling insertion are essentially the same: they follow the same basic procedure as the fixed-point based on-the-fly pointer analysis algorithm (though IPA has small implementation optimizations). We hence do not duplicate the numbers for these approaches because they are mostly the same as IPA. For REVISER, it has similar performance as reported in Table 4 to handle incremental insertions, which is much slower than IPA.

Overall, handling insertion is much faster than handling deletion. For all the benchmarks, *IPA-1* takes only 1ms or less on average to analyze an insertion and 4.1s in the worst case, and *IPA-48* further reduces the worst case to 0.6s. Compared to the exhaustive analysis that takes 2.4h on average, IPA improves the performance by five orders of magnitude or more per insertion. For instance, for *Avrora*,

Table 5. Performance of the incremental algorithms for handling addition.

Benchmark	**Full**	**REVISER**		**IPA-1**		**IPA-48**	
		Avg	**Worst**	**Avg**	**Worst**	**Avg**	**Worst**
avrora	4.8h	134s	248s	0.99ms	1s	0.82ms	0.1s
batik	4.1h	225s	343s	0.86ms	0.8s	0.41ms	0.1s
eclipse	1h	116s	211s	0.74ms	0.4s	0.62ms	0.07s
fop	3.3h	104s	200s	1.33ms	5s	0.82ms	0.3s
h2	3.9h	132s	247s	0.18ms	17s	0.16ms	1.1s
jython	3.2h	-	-	0.49ms	12s	0.35ms	0.9s
luindex	2.9h	124s	245s	1.1ms	9s	0.88ms	1.7s
lusearch	2.5h	113s	217s	0.83ms	0.6s	0.42ms	0.2s
pmd	0.65h	229s	338s	0.61ms	0.2s	0.53ms	0.1s
sunflow	3.5h	216s	337s	0.87ms	7s	0.66ms	2.9s
tomcat	0.6h	113s	224s	0.32ms	0.3s	0.19ms	0.05s
tradebeans	0.75h	102s	211s	0.45ms	0.3s	0.37ms	0.1s
tradesoap	0.8h	107s	214s	0.62ms	0.3s	0.43ms	0.2s
xalan	0.47h	101s	209s	0.9ms	0.4s	0.5ms	0.13s
Average	2.4h	140s	250s	0.72ms	4.1s	0.51ms	0.6s

the exhaustive analysis takes 4.8h, whereas *IPA-1* and *IPA-48* take only 1s and 0.1s respectively in the worst case. The performance results indicate that incremental algorithms are fast enough for practical use in the programming phase with respect to incremental code insertions (but not deletion).

7.3 Analysis of the Special Cases

The results in Tables 4 and 5 show that handling deletion is much slower than handling insertion, and in a few cases a statement deletion can take over 20s for IPA to analyze. For example, for *Jython*, both *IPA-1* and *IPA-48* takes less than 0.1s to analyze an insertion in the worst case, whereas for deletion they take 480s and 22s, respectively. The reasons for such special cases are twofold. First, the algorithm for handling deletion is inherently more complex than that for insertion. For the slow cases, the in and out degrees of a change impacted node are typically very large and the chain of dependent variables is long, which require checking a large number of incoming neighbours and propagating the deletion changes to a large number of outgoing neighbours for each node. Second, the underlying on-the-fly pointer analysis implementation provided in WALA is optimized for handling insertion but not for deletion, since the incremental insertion algorithm follows the same flow as that of the exhaustive pointer analysis for each statement.

However, such special cases are very rare. Table 6 reports the number of special statements in each benchmark of which the root node in the PAG corresponding to the deletion has $M > 1000$ incoming neighbours and of which the deletion requires updating $N > 100$ points-to sets, which typically take IPA much longer to analyze than by the rest of the statments. Overall, the percentage of such special statements over all statements in the program is very small. The maximum percentage of such cases in the benchmarks is only 3‰ (in *Fop*), and the percentage for the majority of the benchmarks is under 1‰. And for $M * N > 100K$, the maximum percentage of such cases is only 1‰. In other words, out of every one hundred statements only one of them can affect a node with more than 1K incoming neighbours, and only three statements can affect more than 100 points-to sets. And only one of every one thousand statements can affect both affect more than 1K incoming neighbours and more than 100 points-to sets, which are expensive to analyze. The majority of the worst cases in each benchmark belong to this small category. For the rest 99.9% cases, they typically take IPA several milliseconds or at most 1s to process.

Table 6. Statistics of the special cases. M: #incoming neighbours of the root node after deleting a statement. N: #updated points-to sets after the change.

Benchmark	**#Total**	**M>1000**	**Avg time**	**N>100**	**Avg time**	**M*N>100K**	**Avg time**
avrora	220104	2785(**1**%)	437ms	419(**2**‰)	1.1s	62(**0.3**‰)	17.7s
batik	171126	1903(**1**%)	173ms	201(**1**‰)	1s	104(**0.6**‰)	10.7s
eclipse	259872	630(**2**‰)	5325ms	89(**0.3**‰)	2s	17(**0.06**‰)	11.6s
fop	87064	12(**0.1**‰)	391ms	2351(**3**%)	2.3s	91(**1**‰)	22s
h2	129628	1607(**1**%)	91ms	228(**2**‰)	1.1s	7(**0.05**‰)	9.3s
jython	862570	31(**0.04**‰)	234ms	218(**0.3**‰)	3.1s	9(**0.01**‰)	37s
luindex	71500	77(**1**‰)	302ms	194(**3**‰)	2.8s	11(**0.2**‰)	8.8s
lusearch	27306	23(**1**‰)	42ms	61(**2**‰)	0.9s	0	0
pmd	131101	2535(**2**%)	138ms	106(**1**‰)	13.8s	95(**0.7**‰)	4.9s
sunflow	37208	27(**0.7**‰)	102ms	30(**0.8**‰)	1.1s	0	0
tomcat	119438	2396(**2**%)	89ms	374(**3**‰)	2.8s	51(**0.4**‰)	7.8s
tradebeans	15832	40(**3**‰)	32ms	12(**0.8**‰)	4.2s	0	0
tradesoap	17988	45(**3**‰)	45ms	9(**0.5**‰)	5.4s	0	0
xalan	36654	590(**1.6**%)	21ms	140(**4**‰)	2s	0.4(**1**‰)	6.3s

Table 7. Performance of incremental race detection. #Slow: statements that take the tool ≥1s.

Benchmark	**Full**	**ECHO**			**ECHO+IPA-48**		
		Insert	**Delete**	**#Slow(‰)**	**Insert**	**Delete**	**#Slow(‰)**
avrora	6.9h	2123ms	54s	431(**2**‰)	2064ms	2.1s	72(**0.3**‰)
batik	6.9h	1313ms	49s	1762(**10**‰)	1272ms	1.3s	203(**1.2**‰)
eclipse	1.2h	341ms	14s	1204(**5**‰)	316ms	0.4s	62(**0.2**‰)
h2	4h	56ms	37s	315(**4**‰)	33ms	0.06s	9(**0.1**‰)
jython	3.3h	33ms	43s	613(**5**‰)	19ms	0.04s	7(**0.1**‰)
luindex	3h	25ms	22s	1280(**2**‰)	5ms	0.04s	37(**0**‰)
lusearch	2.6h	14ms	17s	82(**1**‰)	7ms	0.02s	5(**0.1**‰)
pmd	0.8h	50ms	14s	224(**8**‰)	42ms	0.05s	11(**0.4**‰)
sunflow	3.6h	21ms	47s	215(**2**‰)	1ms	0.04s	2(**0**‰)
tomcat	0.7h	29ms	10s	100(**3**‰)	6ms	0.03s	10(**0.3**‰)
tradebeans	0.8h	8ms	4s	137(**1**‰)	2ms	0.01s	0(**0**‰)
tradesoap	0.9h	10ms	4s	47(**3**‰)	1ms	0.02s	3(**0.2**‰)
xalan	0.5h	4ms	38s	76(**4**‰)	0.7ms	0.01s	0(**0**‰)
Average	2.8h	310ms	27s	**4‰**	290ms	0.3s	**0.2‰**

7.4 Application on Interactive Race Detection

We have also evaluated IPA for improving the performance of interactive race detection implemented in ECHO [88]. The ECHO tool relies on incremental pointer analysis to detect data races incrementally, in which pointer analysis is used to determine the thread shared objects and synchronization locks. We run both the default ECHO (which uses a hybrid of reset-recompute and the reachability-based algorithm for incremental pointer analysis) and the ECHO with IPA configured with a pool of 48 threads (*IPA-48*) for all the 13 multithreaded applications in DaCapo-9.12 (except *Fop* which is single-threaded).

The results are reported in Table 7. Column 2 (*Full*) reports the time taken by the one-shot whole program race detection (*i.e.*, with no incremental analysis). The whole-program race detection takes from half an hour to seven hours to analyze a benchmark (e.g., 6.9h for *Avrora* and *Batik*) and 2.8h on average. Columns 3-4 report the time taken by the original ECHO for a statement insertion and deletion on average, respectively. The time for ECHO includes that taken by the incremental algorithms for updating the PAG and for detecting races per change. With incremental analysis, ECHO is much faster than the whole-program race detection. ECHO takes under 0.3s to process a statement insertion and 27s to process a statement deletion on average, two to three orders of magnitude faster than the whole-program race detection. Column 5 (*#Slow(‰)*) reports the number and the per mille of the slow statements, *i.e.*, those statements that take the tool at least 1s to analyze. For all benchmarks, 1-10‰ of the statements (4‰ on average) require more than 1s by ECHO.

Columns 6-8 report the corresponding data for ECHO with IPA-48. The results show that IPA significantly improves the performance of ECHO, especially for deletion. The original ECHO takes 3s-54s to process each statement detection, whereas *ECHO+IPA-48* takes only 0.01s-2s (0.3s on average), which is 100X faster. Moreover, IPA significantly reduces the number of slow statements. For all the benchmarks, the per mille of slow statements by *ECHO+IPA-48* is under 1.2‰ (0.2‰ on average), which reduces the number of slow statements by the original ECHO by 10X-100X. In other words, with IPA, for 99.9% of the cases ECHO now takes less than 1s to analyze an incremental statement change.

7.5 Discussion on Memory Usage

Because IPA stores the intermediate graphs in memory, for large programs it may require a large memory to run continuously. In particular, in our current implementation we do not remove the PAG and call graph nodes even if they have no incoming and outgoing edges after a statement deletion (in order to improve performance in case the statement is added back later). In our experiments, the largest memory consumption we observed was around 140GB. This may be too excessive for laptops running IDEs, where memory resource is limited. To reduce memory usage, we can periodically clean those graph nodes. Alternatively, we can offload IPA and the client analyses (e.g., data race detection) to remote servers [41] and integrate with a language-independent IDE through the Language Server Protocol (LSP) [1]. We plan to implement IPA in the WALA IDE [36] (which supports LSP) in future work.

8 RELATED WORK

As studied in [27], pointer analysis has been extensively researched along several dimensions, which affect the trade-off between cost and precision, e.g., context-sensitivity [18, 32, 50, 66, 71, 86], flow-sensitivity [13, 24, 25, 39], path-sensitivity [77], field-sensitivity [54, 71], demand-driven [26, 68, 74], algorithmic complexity analysis [15, 34, 65, 73], as well as incremental pointer analysis [4, 31, 60, 63, 79]. In this section, we focus on discussing existing work related to incremental and parallel algorithms and SCC optimizations.

8.1 Incremental Algorithms

Most existing algorithms for handling dynamic program changes assume a static program call graph [4, 31, 60, 63]. Existing incremental algorithms [63, 88] based on reset-recompute and graph reachability are difficult to scale and parallelize, and/or reduce the analysis precision. In particular, they cannot handle code deletion efficiently. In contrast, our new algorithms do not incur any redundant recomputations nor require expensive graph reachability analysis, and are parallelizable without losing precision compared to the exhaustive analysis with the same dimensions.

Sreedhar et al. [70] develop a classic incremental data-flow analysis (but not pointer analysis) based on elimination methods. We refer the readers to [70] for many previous classical incremental analysis.

Saha et al. [60, 61] propose an incremental and demand-driven points-to analysis based on the DRed algorithm [22], which uses a *deletion-rederivation* strategy similar to the reset-recompute algorithm: it marks the affected answers, checks the marked answers, and removes the answers that cannot be rederived. To reduce the redundant checks, a *support graph* is introduced where nodes are the answers, supports and facts, and edges indicate the points-to relations among them. A primary support, which is independent from its answer, is maintained to optimize the marking process. When there is an incremental change, they only mark an answer if its primary support is marked, and then mark all the supports that uses the answer. After the marking stage, if a marked answer has other unmarked supports, they consider the answer is rederived and recursively remove the marks generated from the answer. Otherwise, a recomputation has to be performed on the answer. Instead of PAG, this technique adopts a support graph to represent the points-to relations among variables. However, such a graph requires maintaining primary supports for each answer. Besides, the "mark-check-remove mark" method redundantly propagates the marks, since it cannot recognize whether an answer should be removed at first glance. However, our analysis can discover whether a node need to be updated immediately. Furthermore, our incremental analysis is designed for massively parallelization. However, their rederivation technique requires computing *derivation length* to determine the order in which marked answer should be recomputed first. Such an order prohibits the leverage of parallelization. Besides, the handling of dynamic method calls is very imprecise in this technique: all the methods that have the same number and types of parameters as the call site are considered as targets. In contrast, our new algorithm maintains the call graph and PAG on-the-fly, so that we can easily localize the target methods of a changed method call.

IncA [79] as discussed in Section 3.1 proposes a DSL to express points-to constraints as graph patterns and uses incremental graph pattern matching to update pointer analysis result according to code changes. REVISER [4] proposes an incremental data-flow analysis based on the IFDS/IDE framework. IFDS/IDE is a powerful framework for solving a class of problems with distributive flow functions $f(a) \sqcap f(b) = f(a \sqcap b)$. Because code deletion cannot be modeled by merge operations, IFDS/IDE does not directly apply to incremental pointer analysis. Nevertheless, our work shows that the particular problem of pointer analysis satisfies the “local neighbour” properties (which is not shown by previous research) and that we can leverage them to improve the analysis efficiency. Since these properties are basic graph properties, they are valid beyond pointer analysis and may also be applied in IFDS/IDE for computing distributive problems. It would be interesting to investigate if IFDS/IDE can be adapted to leverage the “local neighbour” properties for pointer analysis.

Incremental computation has been extensively discussed in the domain of datalog evaluation. There are several pointer analyses formulated using datalog frameworks [10, 11, 20, 42, 82]. However, despite intensive research [22, 51] for optimizing the incremental evaluation of datalog, datalog engines are still inefficient to handle incremental deletion of the pointer analysis facts. In our experience with LogicBlox [44], a state-of-the-art datalog engine that supports incremental updates of Datalog facts, it cannot even finish in six hours for handling a statement deletion in our benchmarks.

8.2 Parallel Algorithms

Most existing parallel algorithms are designed to speed up the propagation of initial points-to constraints for whole-program pointer analysis only, which require a static whole program. Differently, our parallel analysis is based on the change idempotency property upon the incremental pointer analysis, which has not been proposed before.

Putta and Nasre [55] propose a parallel replication-based algorithm for pointer analysis: all the initial points-to constraints have been partitioned into n sets and arranged to n threads to propagate points-to sets; each thread has its own copy of conflicting variables and their associated points-to sets; all the copies are merged after the threads have completed their works.

Méndez-Lojo et al. [49] formulate the inclusion-based points-to analysis in terms of graph rewriting rules, which extra constraint edges are added to help the reasoning of points-to relations. By using the Galois system [78], the rules are performed in parallel on non-interfering nodes in the constraint graph. This graph rewriting algorithm has been further implemented on GPU [48] with an efficient graph representation for the constraint graph under the GPU memory model. Nagaraj et al. [52] propose a flow-sensitive pointer analysis which is parallelized based on the graph rewriting rules from [49].

PSEGPT [89] is also designed for parallel flow-sensitive pointer analysis, which relies on a new representation that combines points-to relations and def-use chain on heap. It decomposes the points-to analysis into fine-grained units of work that can be implemented in an asynchronous task-parallel programming model. The operations that propagate the points-to information can be executed in parallel if they obey the data dependence among operations.

Edvinsson et al. [17] discover clusters of points-to constraints that are independent to each other and assign the clusters to different threads, where the independence refers to the true/false branches of a selection and the call targets of a method invocation.

Su et al. [76] propose an inter-query parallelism strategy on the demand-driven CFL-reachability pointer analysis. Each thread fetches a group of queries from a shared work list to perform the computation of points-to sets. In each thread, the order in which queries are processed are determined by *connection distances* to achieve early termination. During the process, shortcut edges are added into the PAG to skip the redundant retraversals of related paths. Our parallelization is intra-query parallelism, which distributes the work performed in computing the related points-to sets of a root pointer among threads.

8.3 SCC optimizations

A number of pointer analysis algorithms [12, 14, 23, 62, 86] have adopted some SCC optimizations to improve performance, *e.g.*, Tarjan's algorithm [81]. However, all these optimizations do not update SCCs according to dynamic graph changes as we do in our incremental algorithm.

La Poutré et al. [33] propose the first algorithm to dynamically maintain the transitive reduction of a directed graph w.r.t. graph edge insertions and deletions, *i.e.*, dynamically collapse/break SCCs. Bergmann et al. [7] adapt the algorithm [33] for incremental graph pattern matching to improve the scalability. Marlowe et al. [47] propose an incremental data flow analysis that can decompose/compose the affected SCCs in the data flow graph whenever there are data flow changes, such that a precise and correct result can be obtained efficiently.

9 CONCLUSION

We have presented the design and implementation of a new incremental pointer analysis, IPA, which significantly improves the scalability of the state-of-the-art. Underpinned by fundamental properties of the on-the-fly Andersen-style pointer analysis, our new algorithms do not incur redundant computations or require expensive graph reachability analysis, and it is parallel. We have implemented our algorithms for Java and integrated our implementation into the open source WALA framework. Our evaluation on a wide range of real-world large complex applications shows that IPA improves the performance of existing algorithms by several orders of magnitude without losing precision. We have also applied IPA for incremental data race detection and shown that IPA significantly improves the performance of a state-of-the-art IDE-based race detector.

10 ACKNOWLEDGEMENTS

We wish to thank Julian Dolby for helping us integrate IPA into WALA and for his invaluable advices on the implementation of incremental pointer analysis. We wish to thank Thomas Ball for insightful discussions on incremental analysis. We also wish to thank Michael Burke and our excellent TOPLAS reviewers for their critical and constructive feedback, which has greatly improved this paper. This work was supported by NSF awards CCF-1552935 and CNS-1617985 and a Google Faculty Research award to Jeff Huang.

REFERENCES

[1] 2018. Language Server Protocol. https://langserver.org/. (2018).

[2] Karim Ali and Ondřej Lhoták. 2012. Application-Only Call Graph Construction. In *Proceedings of the 26th European Conference on Object-Oriented Programming*. 688–712.

[3] Lars Ole Andersen. 1994. *Program Analysis and Specialization for the C Programming Language*. Ph.D. Dissertation.

[4] Steven Arzt and Eric Bodden. 2014. Reviser: Efficiently Updating IDE-/IFDS-based Data-flow Analyses in Response to Incremental Program Changes. In *Proceedings of the 36th International Conference on Software Engineering*. 288–298.

[5] SSA based IR in WALA. 2018. https://github.com/wala/WALA/wiki/Intermediate-Representation-(IR). (2018).

[6] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Robert E. Tarjan. 2016. A New Approach to Incremental Cycle Detection and Related Problems. *ACM Trans. Algorithms* 12, 2 (2016), 14:1–14:22.

[7] Gábor Bergmann, István Ráth, Tamás Szabó, Paolo Torrini, and Dániel Varró. 2012. Incremental Pattern Matching for the Efficient Computation of Transitive Closure. In *Graph Transformations*. 386–400.

[8] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. 169–190.

[9] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. In *Proceedings of the 33rd International Conference on Software Engineering*. 241–250.

[10] Martin Bravenboer and Yannis Smaragdakis. 2009. Exception Analysis and Points-to Analysis: Better Together. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*. 1–12.

[11] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*. 243–262.

[12] Michael G. Burke, Paul R. Carini, Jong-Deok Choi, and Michael Hind. 1994. Flow-Insensitive Interprocedural Alias Analysis in the Presence of Pointers. In *Proceedings of the 7th International Workshop of Languages and Compilers for Parallel Computing*. 234–250.

[13] Arnab De and Deepak D'Souza. 2012. Scalable Flow-sensitive Pointer Analysis for Java with Strong Updates. In *Proceedings of the 26th European Conference on Object-Oriented Programming*. 665–687.

[14] Jens Dietrich, Nicholas Hollingum, and Bernhard Scholz. 2015. Giga-scale Exhaustive Points-to Analysis for Java in Under a Minute. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 535–551.

[15] Jens Dietrich, Nicholas Hollingum, and Bernhard Scholz. 2016. A Note on the Soundness of Difference Propagation. In *Proceedings of the 18th Workshop on Formal Techniques for Java-like Programs*. 3:1–3:5.

[16] LLVM discussions on pointer analysis. 2016. http://lists.llvm.org/pipermail/llvm-dev/2016-March/096851.html. (2016).

[17] Marcus Edvinsson, Jonas Lundberg, and Welf Löwe. 2011. Parallel Points-to Analysis for Multi-core Machines. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*. 45–54.

[18] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. 1994. Context-sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*. 242–256.

[19] Charles L. Forgy. 1982. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artif. Intell.* 19, 1 (1982), 17–37.

[20] Neville Grech and Yannis Smaragdakis. 2017. P/Taint: unified points-to and taint analysis. *PACMPL* 1, OOPSLA (2017), 102:1–102:28.

[21] David Grove and Craig Chambers. 2001. A Framework for Call Graph Construction Algorithms. *ACM Trans. Program. Lang. Syst.* 23, 6 (2001), 685–746.

[22] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. 1993. Maintaining Views Incrementally. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 157–166.

[23] Ben Hardekopf and Calvin Lin. 2007. 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*. 290–299.

[24] Ben Hardekopf and Calvin Lin. 2009. Semi-sparse Flow-sensitive Pointer Analysis. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 226–238.

[25] Ben Hardekopf and Calvin Lin. 2011. 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*. 289–298.

[26] Nevin Heintze and Olivier Tardieu. 2001. Demand-driven Pointer Analysis. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*. 24–34.

[27] Michael Hind. 2001. Pointer Analysis: Haven't We Solved This Problem Yet?. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. 54–61.

[28] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. 1999. Interprocedural Pointer Alias Analysis. *ACM Trans. Program. Lang. Syst.* 21, 4 (1999), 848–894.

[29] Pointer Analysis in WALA. 2017. http://wala.sourceforge.net/wiki/index.php/UserGuide:PointerAnalysis. (2017).

[30] IPA. 2018. https://github.com/april1989/Incremental_Points_to_Analysis. (2018).

[31] George Kastrinis and Yannis Smaragdakis. 2013. Efficient and Effective Handling of Exceptions in Java Points-to Analysis. In *Proceedings of the 22nd International Conference on Compiler Construction*. 41–60.

[32] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid Context-sensitivity for Points-to Analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 423–434.

[33] J. A. La Poutré and J. van Leeuwen. 1988. Maintenance of Transitive Closures and Transitive Reductions of Graphs. In *Proceedings of the International Workshop on Graph-theoretic Concepts in Computer Science*. 106–120.

[34] William Landi and Barbara G. Ryder. 1991. Pointer-induced Aliasing: A Problem Classification. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 93–103.

[35] William Alexander Landi. 1992. *Interprocedural Aliasing in the Presence of Pointers*. Ph.D. Dissertation.

[36] WALA language-independent IDE support. 2018. https://github.com/wala/IDE. (2018).

[37] Ondřej Lhoták. 2002. *Spark: A flexible points-to analysis framework for Java*. Master's thesis. McGill University.

[38] Ondřej Lhoták and Laurie Hendren. 2008. Evaluating the Benefits of Context-sensitive Points-to Analysis Using a BDD-based Implementation. *ACM Trans. Softw. Eng. Methodol.* 18, 1 (2008), 3:1–3:53.

[39] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2011. Boosting the Performance of Flow-sensitive Points-to Analysis Using Value Flow. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. 343–353.

[40] Yue Li, Tian Tan, Yulei Sui, and Jingling Xue. 2014. Self-inferencing Reflection Resolution for Java. In *Proceedings of the 28th European Conference on Object-Oriented Programming*. 27–53.

[41] Bozhen Liu and Jeff Huang. 2018. D4: Fast Concurrency Debugging with Parallel Differential Analysis. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 359–373.

[42] Yanhong A. Liu and Scott D. Stoller. 2009. From Datalog Rules to Efficient Programs with Time and Space Guarantees. *ACM Trans. Program. Lang. Syst.* 31, 6 (2009), 21:1–21:38.

[43] Benjamin Livshits, John Whaley, and Monica S. Lam. 2005. Reflection Analysis for Java. In *Proceedings of the Third Asian Conference on Programming Languages and Systems*. 139–160.

[44] LogicBlox. 2018. LogiQL. http://www.logicblox.com/technology/. (2018).

[45] Yi Lu, Lei Shang, Xinwei Xie, and Jingling Xue. 2013. An Incremental Points-to Analysis with CFL-Reachability. In *Proceedings of the 22nd International Conference on Compiler Construction*. 61–81.

[46] Nick Malone, Kendra Lesser, Meeko Oishi, and Lydia Tapia. 2014. Stochastic Reachability Based Motion Planning for Multiple Moving Obstacle Avoidance. In *Proceedings of the 17th International Conference on Hybrid Systems: Computation and Control*. 51–60.

[47] Thomas J. Marlowe and Barbara G. Ryder. 1990. An Efficient Hybrid Algorithm for Incremental Data Flow Analysis. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 184–196.

[48] Mario Méndez-Lojo, Martin Burtscher, and Keshav Pingali. 2012. A GPU Implementation of Inclusion-based Points-to Analysis. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 107–116.

[49] Mario Méndez-Lojo, Augustine Mathew, and Keshav Pingali. 2010. Parallel Inclusion-based Points-to Analysis. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. 428–443.

[50] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Trans. Softw. Eng. Methodol.* 14, 1 (2005), 1–41.

[51] Boris Motik, Yavor Nenov, Robert Piro, and Ian Horrocks. 2015. Incremental Update of Datalog Materialisation: The Backward/Forward Algorithm. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*. 1560–1568.

[52] Vaivaswatha Nagaraj and R. Govindarajan. 2013. Parallel Flow-sensitive Pointer Analysis by Graph-rewriting. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*. 19–28.

[53] Y.K. Hwang P.C. Chen. 1998. SANDROS: a dynamic graph search algorithm for motion planning. *IEEE Transactions on Robotics and Automation* 4 (1998), 390–403. Issue 3.

[54] David J. Pearce, Paul H.J. Kelly, and Chris Hankin. 2007. Efficient Field-sensitive Pointer Analysis of C. *ACM Trans. Program. Lang. Syst.* 30, 1 (2007).

[55] Sandeep Putta and Rupesh Nasre. 2012. Parallel Replication-based Points-to Analysis. In *Proceedings of the 21st International Conference on Compiler Construction*. 61–80.

[56] G. Ramalingam. 1994. The Undecidability of Aliasing. *ACM Trans. Program. Lang. Syst.* 16, 5 (1994), 1467–1471.

[57] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 49–61.

[58] Barbara G. Ryder. 1983. Incremental Data Flow Analysis. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. 167–176.

[59] Barbara G. Ryder. 2003. Dimensions of Precision in Reference Analysis of Object-oriented Programming Languages. In *Proceedings of the 12th International Conference on Compiler Construction*. 126–137.

[60] Diptikalyan Saha and C. R. Ramakrishnan. 2005. 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*. 117–128.

[61] Diptikalyan Saha and C. R. Ramakrishnan. 2005. Symbolic Support Graph: A Space Efficient Data Structure for Incremental Tabled Evaluation. In *Logic Programming*. 235–249.

[62] Diptikalyan Saha and C. R. Ramakrishnan. 2006. A Local Algorithm for Incremental Evaluation of Tabled Logic Programs. In *Logic Programming*. 56–71.

[63] Lei Shang, Yi Lu, and Jingling Xue. 2012. 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*. 270–273.

[64] Olin Shivers. 1991. *Control-Flow Analysis of Higher-Order Languages*. Ph.D. Dissertation. Carnegie Mellon University.

[65] Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Found. Trends Program. Lang.* 2, 1 (2015), 1–69.

[66] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective Analysis: Context-sensitivity, Across the Board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 485–495.

[67] Robert Smith, Aaron Sloman, and John. Gibson. 1992. POPLOG's Two-level Virtual Machine Support for Interactive Languages. *Research Directions in Cognitive Science Volume 5: Artificial Intelligence.* (1992), 203–231.

[68] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *30th European Conference on Object-Oriented Programming*, Vol. 56. 22:1–22:26.

[69] Vugranam C. Sreedhar, Michael Burke, and Jong-Deok Choi. 2000. A Framework for Interprocedural Optimization in the Presence of Dynamic Class Loading. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. 196–207.

[70] Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. 1998. A New Framework for Elimination-Based Data Flow Analysis Using DJ Graphs. *ACM Trans. Program. Lang. Syst.* 20, 2 (1998), 388–435.

[71] Manu Sridharan and Rastislav Bodík. 2006. Refinement-based Context-sensitive Points-to Analysis for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 387–400.

[72] Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. 2013. Aliasing in Object-Oriented Programming. Chapter Alias Analysis for Object-oriented Programs, 196–232.

[73] Manu Sridharan and Stephen J. Fink. 2009. The Complexity of Andersen's Analysis in Practice. In *Proceedings of the 16th International Symposium on Static Analysis*. 205–221.

[74] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven Points-to Analysis for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. 59–76.

[75] Bjarne Steensgaard. 1996. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 32–41.

[76] Yu Su, Ding Ye, and Jingling Xue. 2014. Parallel Pointer Analysis with CFL-Reachability. In *Proceedings of the Brazilian Conference on Intelligent Systems*. 451–460.

[77] Yulei Sui, Sen Ye, Jingling Xue, and Pen-Chung Yew. 2011. SPAS: Scalable Path-sensitive Pointer Analysis on Full-sparse SSA. In *Proceedings of the 9th Asian Conference on Programming Languages and Systems*. 155–171.

[78] Galois System. 2018. http://iss.ices.utexas.edu/. (2018).

[79] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. 2016. IncA: A DSL for the Definition of Incremental Program Analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 320–331.

[80] Tian Tan, Yue Li, and Jingling Xue. 2017. Efficient and Precise Points-to Analysis: Modeling the Heap by Merging Equivalent Automata. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 278–291.

[81] Robert Tarjan. 1972. Depth first search and linear graph algorithms. *SIAM JOURNAL ON COMPUTING* 1, 2 (1972).

[82] K. Tuncay Tekle and Yanhong A. Liu. 2016. Precise complexity guarantees for pointer analysis via Datalog with extensions. *TPLP* 16 (2016), 916–932.

[83] Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. 2015. EMF-IncQuery: An integrated development environment for live model queries. *Sci. Comput. Program.* 98 (2015), 80–99.

[84] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*. 13–.

[85] WALA. 2017. T. J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net/. (2017).

[86] John Whaley and Monica S. Lam. 2004. Cloning-based Context-sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 131–144.

[87] Jyh-shiarn Yur, Barbara G. Ryder, and William A. Landi. 1999. An Incremental Flow- and Context-sensitive Pointer Aliasing Analysis. In *Proceedings of the 21st International Conference on Software Engineering*. 442–451.

[88] Sheng Zhan and Jeff Huang. 2016. ECHO: Instantaneous in Situ Race Detection in the IDE. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 775–786.

[89] Jisheng Zhao, Michael G. Burke, and Vivek Sarkar. 2018. Parallel Sparse Flow-sensitive Points-to Analysis. In *Proceedings of the 27th International Conference on Compiler Construction*. 59–70.