AN INSTANTANEOUS FRAMEWORK FOR CONCURRENCY BUG DETECTION

A Thesis

by

BOZHEN LIU

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Shaoming Huang
Committee Members,	Riccardo Bettati
	Peng Li
Head of Department,	Dilma Da Silva

December 2018

Major Subject: Computer Engineering

Copyright 2018 Bozhen Liu

ABSTRACT

Concurrency bug detection is important to guarantee the correct behavior of multi-thread programs. However, existing static techniques are expensive with false positives, and dynamic analyses cannot expose all potential bugs.

This thesis presents an ultra-efficient concurrency analysis framework, D4, that detects concurrency bugs (e.g., data races and deadlocks) "instantly" in the programming phase. As developers add, modify, and remove statements, the changes are sent to D4 to detect concurrency bugs on-the-fly, which in turn provides immediate feedback to the developer of the new bugs. D4 includes a novel system design and two novel parallel incremental algorithms that embrace both change and parallelization for fundamental static analyses of concurrent programs. Both algorithms react to program changes by memoizing the analysis results and only recomputing the impact of a change in parallel without any redundant computation. Our evaluation on an extensive collection of large real-world applications shows that D4 efficiently pinpoints concurrency bugs within 10ms on average after a code change, several orders of magnitude faster than both the exhaustive analysis and the state-of-the-art incremental techniques.

ACKNOWLEDGMENTS

There are a number of people without whom this thesis might not have been written, and to whom I am greatly indebted.

I would like to thank my adviser, Jeff Huang, for giving me the support, comments and encouragement I've desperately needed during in pursuance of my degree. As a beginner in computer science, it took a long haul, with many detours along the way, but he never wavered. His advises and guidance have greatly influenced my approach to computer science research.

Thanks to my other committee members, Riccardo Bettati and Peng Li, for contributing a great deal to my development as a graduate by giving me the benefit of their times and advises.

Thanks to Julian Dolby for helping me with my first project and directing me about the integration procedure.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis committee consisting of Professor Jeff Huang and Riccardo Bettati of the Department of Computer Science and Engineering and Professor Peng Li of the Department of Electrical and Computer Engineering.

All the work conducted for the thesis was completed by the student independently.

Funding Sources

Graduate study was supported by an assistantship from Texas A&M University.

NOMENCLATURE

PTA	Points-to Analysis
pts	points-to set
PAG	Pointer Assignment Graph
PIPTA	Parallel Incremental Points-to Analysis
D4	Distributed Data race & Deadlock Detection
SHB	Static Happens-before
LD	Lock-dependency

TABLE OF CONTENTS

Page

ABSTRACT ii

ACKNOWLEDGMENTS iii

CONTRIBUTORS AND FUNDING SOURCES iv

NOMENCLATURE v

TABLE OF CONTENTS vi

LIST OF FIGURES ix

LIST OF TABLES x

1. INTRODUCTION 1

1.1 Background 1
1.1.1 Pointer Analysis 2
1.1.2 Happens-before Analysis 3
1.2 Thesis Contribution 3
1.3 Outline Of Thesis 4

2. PARALLEL INCREMENTAL POINTS-TO ANALYSIS 5

2.1 Background 5
2.2 Introduction To Points-to Analysis 6
2.3 Performance Bottleneck In Handling Deletion 9
2.3.1 Reset-recompute Algorithm 10
2.3.2 Reachability-based Algorithm 12
2.3.3 Graph Pattern Matching 12
2.4 Observed Properties 13
2.5 Basic Incremental Algorithm 17
2.5.1 Incremental SCC Detection 19
2.5.2 Incremental Edge Deletion 20
2.5.3 Incremental Edge Addition 23
2.6 Parallel Incremental Algorithm 24
2.6.1 Synchronization-free Implementation 27

2.7 End-to-end Incremental Points-to Analysis ... 29
2.7.1 Adapting To Context-sensitive, Flow-sensitive And Other Problems 32
2.7.1.1 Context-sensitive ... 32
2.7.1.2 Flow-sensitive ... 33
2.7.1.3 Other Problems ... 34
2.7.2 Scheduling Of Changed Statements ... 34
2.8 Related Works And Comparison ... 35
2.8.1 Incremental Algorithms ... 35
2.8.2 Parallel Algorithms ... 37
2.8.3 SCC Optimizations ... 39

3. PARALLEL INCREMENTAL HAPPENS-BEFORE ANALYSIS ... 40

3.1 Background ... 40
3.2 Related Works ... 41
3.3 An Example ... 41
3.4 Limitations In Existing Techniques ... 42
3.5 A New SHB Graph Construction ... 44
3.6 The New SHB Graph Update For Incremental Changes ... 45
3.7 How To Compute The Happens-before Relation ... 46

4. D4: A FAST CONCURRENCY DEBUGGING FRAMEWORK ... 48

4.1 Background ... 48
4.2 Related Works ... 49
4.3 Change Extraction ... 49
4.4 Data Race Detection ... 51
4.5 Deadlock Detection ... 52
4.5.1 Lock-dependency Graph ... 52
4.5.2 Deadlock Detection ... 52
4.5.3 LD Graph Update For Incremental Changes ... 53
4.5.4 Incremental Deadlock Detection ... 54
4.6 The Distributed System Design ... 54
4.6.1 Parallel Analysis Framework ... 55
4.6.2 Graph Storage ... 56
4.6.3 Message Format ... 56

5. EVALUATION ... 57

5.1 Evaluation Methodology ... 57
5.2 Benchmarks ... 58
5.3 Evaluation Of PIPTA ... 58
5.4 Evaluation Of D4 ... 59
5.4.1 Performance ... 59

5.4.2 Precision 62
6. CONCLUSION 64
REFERENCES 66

LIST OF FIGURES

FIGURE Page

2.1 An example of the PAG. 7

2.2 An example of an edge deletion in the PAG. 11

2.3 Illustration of the incoming neighbours property. 14

2.4 Illustration of the outgoing neighbours property. 16

2.5 Three SCC scenarios. 18

2.6 An example of parallel change propagation. 27

3.1 An Example for SHB Analysis. 41

3.2 The SHB graph for the example in Figure 3.1. 43

3.3 The new SHB graph for the example in Figure 3.1. 46

4.1 An example for the LD graph construction. 53

LIST OF TABLES

TABLE Page

2.1 Andersen's Constraints for Java. 8

2.2 Java statements and their corresponding PAG edges. 30

3.1 Nodes in the SHB Graph.... 42

3.2 Edges in the SHB Graph 42

3.3 Edges in the New SHB Graph 44

5.1 Benchmarks and PAG metrics. 58

5.2 Performance of Exhaustive and Existing Incremental Pointer Analysis Algorithms. 60

5.3 Performance of Exhaustive and New Incremental Pointer Analysis Algorithms. 60

5.4 Performance of Concurrency Bug Detection. 61

5.5 Results of Detected Concurrency Bugs. 63

1. INTRODUCTION

1.1 Background

Writing correct parallel programs is notoriously challenging due to the complexity of concurrency and the non-deterministic property. Concurrency bugs, such as data races and deadlocks, are easy to introduce but difficult to detect and fix, especially for real-world applications with large code bases. Most existing techniques [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11] either miss many bugs or cannot scale. A common limitation is that they are mostly designed for *late phases* of software development, such as testing or production. Consequently, it is hard to scale these techniques to large software because the whole code base has to be analyzed. Moreover, it may be too late to fix a detected bug, or too difficult to understand a reported warning, because the developer may have forgotten the coding context to which the warning pertain.

A promising way to detect and fix concurrency bugs is during the programming phase, so that the bugs can be fixed as soon as possible by providing developers early feedback, and the expensive analyses can be amortized by analyzing a portion of the whole code bases. To design such an efficient detection, it requires expensive static analyses, such as pointer analysis and happens-before analysis, which can take minutes or hours for real-world programs.

ECHO [12] is such an IDE-based technique that detects data races incrementally in the programming phase. Upon a change in the source code (addition, deletion or modification), instead of exhaustively re-analyzing the whole program, it analyzes the change only and recompute the impact of the change for race detection by memorizing the intermediate analysis results. This not only provides early feedback to developers in order to reduce the cost of bugs, but also enables efficient bug detection by amortizing the analysis

cost. An incremental pointer analysis and a static happens-before analysis contribute to the fast response of race detection. As reported in [12], for 92+% of the code changes in small/middle size programs, ECHO takes no more than 0.1s to detect races.

However, the scalability of the techniques behind ECHO is poor for large real-world programs, because its incremental analysis as well as the static happens-before analysis can still be too slow to be applied in the programming phase. In addition, ECHO runs entirely in the same process as the IDE, which severely limits its performance due to the limited CPU and memory resources. Further, ECHO can only detect races but not any other concurrency bugs such as deadlocks. The ability to detect deadlocks is particularly important for ECHO-like race detection tools, because once a data race is detected, programmers often use locks to fix the race, which may well introduce new deadlock bugs.

1.1.1 Pointer Analysis

Pointer analysis, or points-to analysis, is a fundamental program analysis that reasons about the value of pointer variables statically, i.e., what memory locations or objects can a pointer point to or a variable reference at runtime? There are many dimensions of precision that can be modeled when approximating pointer analysis, such as flow-sensitivity, context-sensitivity, field-sensitivity, the heap model, representation of pointer information, branch conditions and array indexing. The area of pointer analysis has been the focus of intensive research [13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26], because virtually all interesting static program analyses rely on pointer analysis, e.g., error-finding, model checking, systems hardening, security analysis, etc. By improving the precision and scalability of pointer analysis, we can directly contribute to the effectiveness of a wide variety of program analyses that list above.

In this thesis, we focus on the classical Andersen's analysis [13] of the Java-like object-oriented language, which is context/flow-insensitive and object/field-sensitive.

1.1.2 Happens-before Analysis

Happens-before analysis is also a fundamental program analysis, which computes the happens-before relation between two events, for example, one event should happen before or after or concurrently with another event. The happened-before relation is formally defined as the least strict partial order on events, which is transitive, irreflexive and antisymmetric. This is a maturely studied researched area with extensive reference, such as the widely used *vector clock* from Lamport timestamps [27], and the later developed techniques [3, 12, 28]. The scalability of these techniques still have a large room to improve, which can benefit many program analysis applications, especially the concurrency bug detection

1.2 Thesis Contribution

This thesis makes the following contributions:

- A novel end-to-end parallel incremental points-to analysis approach, PIPTA, that dramatically improves the performance and practicality of pointer analysis without losing precision. In particular, we develop a novel incremental algorithm that efficiently handles code deletions without any change impact recomputation nor the expensive graph reachability analysis.

- A new parallel incremental algorithms of static happens-before analysis, for efficiently analyzing concurrent programs by exploiting both the change-centric nature of programming and the algorithmic parallelization of fundamental static analyses.

- A design and implementation of an ultra-efficient interactive concurrency analysis framework, D4, that detects data races and deadlocks instantly in the IDE, *i.e.*, in tens of milliseconds on average as they are introduced into the program.

Besides, PIPTA has been integrated into the popular Java program analysis framework, WALA [29]. The paper of the static framework [30] is accepted by PLDI'18.

1.3 Outline Of Thesis

Chapter 2 describes the new parallel incremental points-to analysis, PIPTA. The chapter gives the background of the basic Andersen's analysis, the performance bottlenecks in related works, and then describes our discoveries and algorithms.

Chapter 3 describes the new parallel incremental static happens-before analysis, which includes the weakness of related works and the new graph representation.

Chapter 4 describes the data race and deadlock detection implemented in D4. This chapter introduces related works, the lock-dependency graph we adopted for the whole program and incremental deadlock detection.

Chapter 5 describes the evaluation of PIPTA for improving the scalability of incremental points-to analysis and our framework for data race and deadlock detection for improving the efficiency of concurrency bug detection.

Finally, Chapter 6 concludes the thesis by summarize the thesis, recapping our contributions and listing future works.

2. PARALLEL INCREMENTAL POINTS-TO ANALYSIS [1]

This chapter presents the background information of Andersen's points-to analysis, discuss the related works and illustrate our parallel incremental points-to analysis (PIPTA) which can scale to large programs. Section 2.1 illustrates the importance in studying points-to analysis and the main challenges. Section 2.2 introduces the basic information in points-to analysis and incremental points-to analysis. Section 2.3 presents why the existing techniques of incremental points-to analysis are inefficient. Section 2.4 illustrates the new properties inferred from our observations on the PAG, which are the foundations of our new algorithms. Section 2.5, 2.6 and 2.7 explains the detailed algorithms of PIPTA. Section 2.8 lists the related works and compares them with PIPTA.

2.1 Background

Like many other static analyses, a precise pointer analysis is undecidable [31, 32]. Thus, all pointer analysis techniques approximate the results and aim to improve the precision and/or efficiency. A key challenge in pointer analysis, however, is how to scale to large real-world programs without sacrificing the precision too much. The complexity of the state-of-the-art inclusion-based pointer analysis [13] is cubic in the program size, which is inherently slow for large programs. According to the most recent developments [14, 21, 33], for real-world programs with hundreds of thousand of lines of source code, the context-insensitive, field-sensitive pointer analysis requires a couple of minutes to compute, and the more-precise context-sensitive analysis requires tens of minutes or even hours. Partly due to this computational challenge, such reasonably-precise pointer

[1]Reprinted with permission from "D4: Fast Concurrency Debugging with Parallel Differential Analysis" by Bozhen Liu, Jeff Huang, 2018. Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, 359-373, Copyright [2018] by Association for Computing Machinery, Inc. Reprinted by permission.

analyses are not widely used in production compilers, missing many potential optimization opportunities.

One promising solution to this challenge is incremental pointer analysis, which memoizes the analysis results and only recomputes the impact after a change. Incremental pointer analysis is particularly useful for applications in which the analysis has to run repeatedly with respect to frequent but small program changes, *e.g.*, bug finding in the programming phase [12] and incremental compilation [34], because analyzing small changes is often much faster than rerunning the pointer analysis for the entire program.

Researchers have investigated and developed several incremental algorithms, which show significant performance improvements over the exhaustive pointer analysis. However, incremental pointer analysis still faces several major challenges in applying to large real-world programs. First, existing incremental algorithms (albeit fast in simple cases) are still too slow in many complex cases, especially for handling code deletions. Second, most existing algorithms assume a pre-built call graph of the program, which does not hold for scenarios where the call graph itself can be changed by the code updates. Third, some algorithms do not preserve precision but compute a less precise points-to result than the exhaustive analysis.

2.2 Introduction To Points-to Analysis

Points-to analysis is to determine the set of heap locations or objects that a variable x can point to during execution, which is the points-to set of x, denoted by $pts(x)$. Since our targets are Java programs, we will focus on Java-like object-oriented language from now.

Points-to analysis is often cast as a graph transitive closure problem, such as pointer assignment graph[35] and flow graph [36]. Here, we adopt the *pointer assignment graph* (PAG) in our analysis, where the nodes in the PAG represent for program variables, abstract locations and fields of abstract locations, and each edge represents the subset-based points-

to constraint between two nodes.

Andersen's analysis, a.k.a. inclusion-based analysis, is the most precise of the context- and flow-insensitive points-to analysis. In Andersen's analysis, the edges represent subset constraints between points-to sets, *i.e.*, an edge $o_1 \rightarrow x$ means that $o_1 \in pts(x)$, and an edge $x \rightarrow y$ means that $pts(x) \subseteq pts(y)$. Figure 2.1 shows an example of PAG for Java programs, in which there are three kinds of nodes: variable nodes (i.e., *x* and *y*), field node (*i.e.*, $x.f$) and object nodes (*i.e.*, o_1 and o_2). Their points-to sets are $pts(x) = o_1$ and $pts(y) = o_1, o_2$.

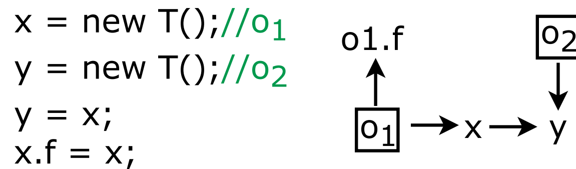

Figure 2.1: An example of the PAG.

There are six essential types of statements considered in Andersen's analysis, which are listed from ① to ⑥ in Table 2.1 with their corresponding points-to constraints and points-to edges. For each type of statements, the points-to sets of the related variables are updated as follows:

① *x* = *new T()*: add a new object o_i to $pts(x)$.

② *x* = *y*: add $pts(y)$ to $pts(x)$.

③ *x* = *y.f* : for each object $o_i \in pts(y)$, add $pts(o_i.f)$ to $pts(x)$.

④ *x.f* = *y*: for each object $o_i \in pts(x)$, add $pts(y)$ to $pts(o_i.f)$.

⑤ *x = y[i]* : for each object $o \in pts(y)$, add $pts(o.*)$ to $pts(x)$.

⑥ *x[i] = y*: for each object $o \in pts(x)$, add $pts(y)$ to $pts(o.*)$.

*f is an object field.

Array accesses (⑤ and ⑥) are considered as field accesses to a constant field by ignoring the array index. Different array indices are not distinguished but represented by a special constant "$*$". We can have a more precise result by create variable nodes for the elements represented by different array indexes, which is expensive.

For points-to analysis in Java programs, adopting field-sensitive is important to improve the precision, which treats each instance field of each abstract location separately.

Statements	Points-to Constraints	PAG Edges
① $x = new\ T()$	$o_T \in pts(x)$	$o_T \to x$
② $x = y$	$pts(y) \subseteq pts(x)$	$y \to x$
③ $x = y.f$	$\forall o \in pts(y)$: $pts(o.f) \subseteq pts(x)$	$\forall o \in pts(y)$: $o.f \to x$
④ $x.f = y$	$\forall o \in pts(x)$: $pts(y) \subseteq pts(o.f)$	$\forall o \in pts(x)$: $y \to o.f$
⑤ $x = y[i]$	$\forall o \in pts(y)$: $pts(o) \subseteq pts(x)$	$\forall o \in pts(y)$: $o \to x$
⑥ $x[i] = y$	$\forall o \in pts(x)$: $pts(y) \subseteq pts(o.i)$	$\forall o \in pts(x)$: $y \to o.i$
⑦ $x = z.m(y)$*	$pts(r) \subseteq pts(x)$, $pts(y) \subseteq pts(p)$	$r \to x$, $y \to p$

Table 2.1: Andersen's Constraints for Java.

Even though the basic Andersen's algorithm is precise enough for most applications, it requires a pre-built call graph, which does not work for the incremental scenario: we must handle dynamic changes to the call graph due to the method-invoke statement.

On-the-fly Andersen's algorithm [15, 20, 36] can process dynamic call graph changes as well as PAG changes. Algorithm 1 outlines the flow of the on-the-fly algorithm, which implements a worklist algorithm. The algorithm starts with only the initial reachable

methods (*e.g.*, the main method), from which an initial set of points-to constraints are constructed. In each iteration, an input set of constraints are consumed and potentially a new set of constraints are produced, which are again consumed by the next iteration. Specifically, each iteration consists of two steps:

(1) Evaluates the input constraints by following the Andersen's algorithm in Table 2.1 to compute the points-to set of each variable and generates new constraints if necessary;

(2) Resolves 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 extracts new constraints for the newly discovered method calls.

The union of the new constraints from these two steps is then provided to the next iteration as the input constraints. This procedure continues until a fixed point is reached.

Algorithm 1: On-the-fly pointer analysis

```
  // Δ:  the new constraints in each iteration
1 Δ ← initial points-to constraints;
  // repeat until Δ is empty
2 while Δ ≠ ∅ do
      // consume Δ and return new constraints
3     Δ1 = runAndersonsAnalysis(Δ)
4     Δ2 = extractNewMethodCallConstraints(Δ1)
5     Δ = Δ1 ∪ Δ2
6 end
```

2.3 Performance Bottleneck In Handling Deletion

Incremental pointer analysis must handle two basic types of changes: adding a statement and deleting a statement, because any code change can be composed from one or

more additions and/or deletions.

Handling addition is mostly straightforward based on the on-the-fly Andersen's points-to analysis described in the previous section. Specifically, 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 in Algorithm 1.

However, handling deletion is much more complicated than addition. Intuitively, it is the reverse of addition and, if we can track the state changes of the PAG by each addition statement, we may undo the changes after deleting the same statement. This intuition is incorrect because changes are often dependent on each other. For example, consider three consecutive code changes: adding a statement $b = a$, adding another statement $c = b$, and then deleting the first statement $b = a$. When $b = a$ is added, the points-to set of b is updated to include those in $pts(a)$, *i.e.*, $pts(b)$= $pts(b) \cup pts(a)$. When $c = b$ is added, 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 three categories of approaches proposed in the existing literature for handling deletion: *reset-recompute* [12, 37, 38, 39, 40], *reachability-based* [12, 41, 42] and *graph pattern matching* [43]. However, all approaches suffer from performance limitations, are difficult to scale to large programs and to parallel the algorithms.

2.3.1 Reset-recompute Algorithm

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, upon a deletion, one can first remove from the PAG all edges related to

the deleted statement and reset (set to empty) the points-to sets of their destination nodes as well as all nodes that they 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.

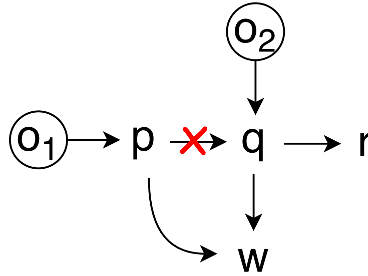

Figure 2.2: An example of an edge deletion in the PAG.

Consider an example in Figure 2.2, in which an edge $p \to q$ is deleted from the PAG (e.g., due to the deletion of a statement $q = p$ in the program). The root variable of the change is q, since its points-to set may be changed immediately because of the edge deletion. The reset-recompute algorithm first resets $pts(q)$ as well as $pts(w)$ and $pts(r)$ to empty (because r and w are reachable from q). Then it extracts the points-to constraints $pts(q)$ = $pts(q) \cup o_2$, $pts(r)$= $pts(r) \cup pts(q)$, $pts(w)$= $pts(w) \cup pts(q)$, and $pts(w)$= $pts(w) \cup pts(p)$, from the four edges connected to the three reset nodes, *i.e.*, $o_2 \to q$, $q \to r$, $q \to w$ and $p \to w$, and recomputes $pts(q)$, $pts(z)$ and $pts(w)$ until reaching a fixed point. The final values of the points-to sets are: $pts(p)$={o_1}, $pts(q)$={o_2}, $pts(r)$={o_2} and $pts(w)$={o_1, o_2}.

The reset-recompute algorithm is inefficient because most computations on the points-to sets of the reset nodes could be redundant. For example, both before and after the deletion, $pts(w)$ remains the same and o_2 is included the points-to sets of q, r and w.

2.3.2 Reachability-based Algorithm

The basic idea of the reachability-based algorithm [12] 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 repetitive whole-graph reachability analysis, which is expensive for large PAGs.

Consider again the example in Figure 2.2. Upon the deletion of the edge $p \rightarrow q$, the algorithm first checks if p is still reachable to q (*i.e.*, via another path without $p \rightarrow q$). If yes, then the algorithm stops with no changes to any points-to set. Otherwise, it goes on to check if any object in $pts(p)$ should be removed from $pts(q)$, by checking if the corresponding object node can reach q in the PAG. In this case, $pts(p)$ contains only o_1 which cannot reach q, hence o_1 is removed from $pts(q)$. Because $pts(q)$ is changed, the algorithm then continues to propagate the change by checking the nodes connected to q (*i.e.*, r and w). Finally, because o_1 cannot reach r but can reach w (via the path $o_1 \rightarrow p \rightarrow w$), o_1 is removed from $pts(r)$ but $pts(w)$ remains unchanged.

The main scalability bottleneck of the reachability-based algorithm is that the worst case time complexity for checking the path reachability is linear in the PAG size, which can be very large for real-world programs.

2.3.3 Graph Pattern Matching

A domain-specific language (DSL) is designed by IncA [43] to incrementally update program analysis result, which is independent of the analyzed program languages. IncA 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). Taking an AST of a program as input, pattern functions either reject the input or compute its corresponding graph patterns to express the relations between AST nodes (*i.e.*, program statements).

The points-to analysis in IncA is built on Andersen's algorithm, which uses a relation $PointsTo(x, y)$ to represent that variable x can point to variable y. There is no abstraction of memory allocation in IncA, so this analysis is actually an alias analysis that tells whether two variables can refer to the same storage location. Besides, this analysis is flow-sensitive, which is built on the incremental control-flow analysis from IncA.

For incremental statement changes, IncA performs an incremental graph pattern matching adopted from EMF-IncQuery [44] to propagate changes to all dependent points-to relations and reanalyze the changed program entities. EMF-IncQuery is an incremental graph query engine to capture and execute live queries over models, which is highly scalable over large programs.

According to the paper, the incremental analysis can complete the update within tens of milliseconds, which is efficient when comparing with its whole program analysis (terminates in seconds). But the evaluated benchmarks only contains $<$50 KLOC. Hence, the millisecond performance cannot be guaranteed on large benchmarks (50 KLOC $\sim$ 1 MLOC).

2.4 Observed Properties

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 a consistency property of change propagation in pointer analysis, which allows to parallel the incremental algorithm.

According to Andersen's analysis rules in Table 2.1, 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 edge additions/ deletions in Section 2.5. 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 $r \to q$) 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 2.3. First, because $o \in pts(r)$, due to transitivity, o can reach r. Second, because the PAG is acyclic, there cannot exist a path $o \to \ldots \to q \to \ldots \to r \to q$ (which contains a cycle). $\square$

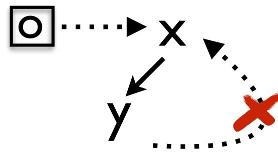

Figure 2.3: Illustration of the incoming neighbours property.

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

Theorem 1: *Suppose an edge $p \to q$ 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 $p \to q$. Because $r \to q$, o can hence reach q without the edge $p \to q$. Therefore, o remains in $pts(q)$ after deleting $p \to q$. 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 2.2. Upon deleting the edge $x \to y$, 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 $q \to w$) 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 2.4. There must exist such a path $o \to \ldots \to r \to w$ from o to w, because $o \in pts(r)$ and $r \to w$. The path may or may not contain q. However, if it contains q, then it must be $o \to \ldots \to q \to \ldots \to r \to w$, which means that q can reach r. It cannot be $o \to \ldots \to r \to \ldots \to q \to w$, because r cannot reach q. □

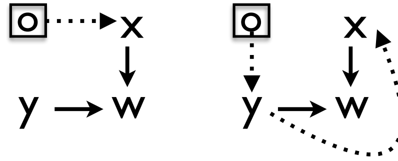

Figure 2.4: Illustration of the outgoing neighbours property.

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

Theorem 2: *Suppose an edge $p \to q$ 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. $\square$

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 2.2. 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.

2.5 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 [45].

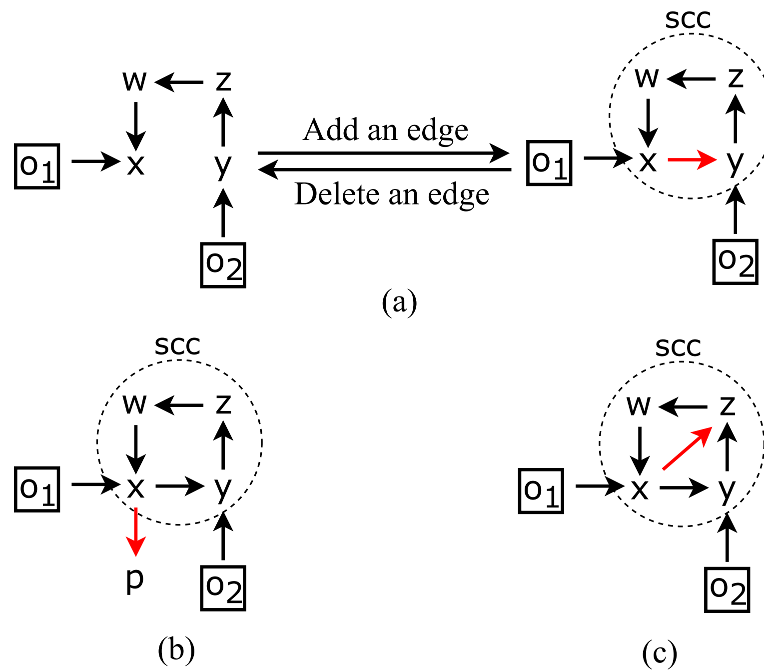

Figure 2.5: Three SCC scenarios.

However, in the incremental setting, the SCCs must be dynamically updated. We first give a brief overview of our incremental SCC detection algorithm, which shared the same main idea with [46, 47]. 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 addition 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.

2.5.1 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 [48] 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 [49] 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 2.5 illustrates the incremental SCC detection with examples. Figure 2.5(a) shows that adding the edge $x \rightarrow y$ creates a new SCC and deleting the edge breaks the SCC. Figure 2.5(b) shows that the edge $x \rightarrow p$ does not belong to any SCC, so adding/deleting the edge does not create new SCCs or affect existing SCCs. Figure 2.5(c) shows that the edge $x \rightarrow z$ belongs to an SCC, but adding/deleting it does not augment or break the SCC.

2.5.2 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 2.5.1. 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 $x \rightarrow y$, 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

Algorithm 2: DeleteEdge(e)

```
   Input  : e - a deleted edge
            pag - the PAG
 1 WL ← e // initialize worklist to e
 2 while WL ≠ ∅ do
 3 |  e ← RemoveOneEdgeFrom(WL)
 4 |  pag ← pag \ {e}
 5 |  DetectSCC(e)
   |  // let e be x → y
 6 |  PropagateDeleteChange(pts(x), y)
 7 end

 8 PropagateDeleteChange(Δ, y):
   Input  : Δ - a set of points-to set changes
            y - a node that Δ propagates to
 9 foreach z → y do //z is an incoming neighbour of y
   |  // Objects in Δ but not in pts(z)
10 |  Δ = Δ \ (Δ ∩ pts(z))
11 |  if Δ = ∅ then
12 |  |  return
13 |  end
14 end
   // remove Δ from pts(y)
15 pts(y) ← (pts(y) \ Δ)
16 foreach y → w do //w is an outgoing neighbour of y
17 |  PropagateDeleteChange(Δ, w)
18 end
19 WL ← CheckNewEdges(Δ, y)
```

run the procedure *CheckNewEdges* (Algorithm 3) 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 affected PAG edges to the worklist (*i.e.*, the edges should be deleted/added). Note that the complex statements (*i.e.*, load, store and call) can introduce new edges. Now, we are processing PAG edge deletion. In *CheckNewEdges*, for each object $o \in \Delta$, and for each node $o.f$ in the PAG that is generated from $y.f$, we include all edges from/to $o.f$ to $list$ (because the node $o.f$ should

Algorithm 3: CheckNewEdges(Δ, y)

```
   Input  : Δ - a set of change
            y - the target node PAG
   Output: list - a list of PAG edges
 1 foreach o ∈ Δ do
 2     foreach Load x = y.f do
           // add its corresponding edge to WL
 3         list ← e  // let e be o.f → x
 4     end
 5     foreach Store y.f = x do
           // add its corresponding edge to WL
 6         list ← e  // let e be x → o.f
 7     end
 8     foreach Call y.m() from m'() do
           // from DeleteEdge:  remove call graph edges from
              caller m'() to callee m()
 9         cg ← AnalyzeDeletedMethod(m', m)
           // from AddEdge:  add call graph edges from caller
              m'() to callee m()
10         cg ← AnalyzeNewMethod(m', m)
           // from DeleteEdge and AddEdge:  add its
              corresponding edges in o.m to list
11         list ← {c → p, r → a}
12     end
13 end
14 return list
```

be removed). For a deleted method call $a = b.m(c)$ (line 9), we remove the caller-callee relation from call graph. Then, we include the edges $c \rightarrow p$ and $r \rightarrow a$ to $list$ (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.

Algorithm 4: AddEdge(e)

```
  Input  : e - an inserted edge
           pag - the PAG
1  WL ← e // initialize worklist to e
2  while WL ≠ ∅ do
3      e ← RemoveOneEdgeFrom(WL)
4      pag ← pag ∪ {e}
5      DetectSCC(e)
       // let e be x → y
6      PropagateAddChange(pts(x), y)
7  end

8  PropagateAddChange(Δ, y):
   Input  : Δ - a set of changes
            y - a node that Δ propagates to
   // Objects in Δ but not in pts(y)
9  Δ = Δ \ (Δ ∩ pts(y))
10 if Δ ≠ ∅ then
       // add Δ to pts(y)
11     pts(y) ← (pts(y) ∪ Δ)
12     foreach y → w do //w is an outgoing neighbour of y
13         PropagateAddChange(Δ, w)
14     end
15     WL ← CheckNewEdges(Δ, y)
16 end
```

2.5.3 Incremental Edge Addition

Algorithm 4 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 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 *CheckNewEdges* procedure has been called again to collect the PAG edges that should be added. The difference from processing edge deletion is that when process a *call* statement $y.m()$ (line 10), it adds the corresponding method call edges, and also analyze the method body if $T.m()$ is new.

2.6 Parallel Incremental Algorithm

Our parallel incremental algorithms are based on a strong *change consistancy* property of our basic incremental algorithms described in Section 2.5.

Lemma 3: Change consistancy property: For an edge addition 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: $p \rightarrow \ldots \rightarrow r_1 \rightarrow q$ ($path_1$) and $p \rightarrow \ldots \rightarrow r_2 \rightarrow q$ ($path_2$), respectively, where p is the root change node (the addition 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 5: ParallelPropagateDeleteChange(Δ, y)

Input : Δ - a set of changes
y - a node that Δ propagates to

```
1  foreach z → y do
2  |   Δ = Δ \ (Δ ∩ pts(z))
3  |   if Δ = ∅ then
4  |   |   return
5  |   end
6  end
7  pts(y) ← (pts(y) \ Δ)
   // all outgoing edges in parallel
8  Parallel foreach y → w do
9  |   ParallelPropagateDeleteChange(Δ, w)
10 end
11 sync {WL} ← CheckNewEdges(Δ, y)
```

Algorithms 5 and 6 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 5 and line 4 in Algorithm 6). 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 2.6 shows an example.

Algorithm 6: ParallelPropagateAddChange(Δ, y)

```
  Input  : Δ - a set of changes
           y - a node that Δ propagates to
1 Δ = Δ \ (Δ ∩ pts(y))
2 if Δ ≠ ∅ then
3 |  pts(y) ← (pts(y) ∪ Δ)
  |  // all outgoing edges in parallel
4 |  Parallel foreach y → w do
5 |  |  ParallelPropagateAddChange(Δ, w)
6 |  end
7 |  sync {WL} ← CheckNewEdges(Δ, y)
8 end
```

Initially, $pts(y) = pts(q) = \{o_1\}$ and $pts(p) = pts(z) = pts(w) = \{o_1, o_2\}$. After deleting the edge $x \rightarrow y$, $pts(y)$ 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). Based on Algorithm 5, 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.

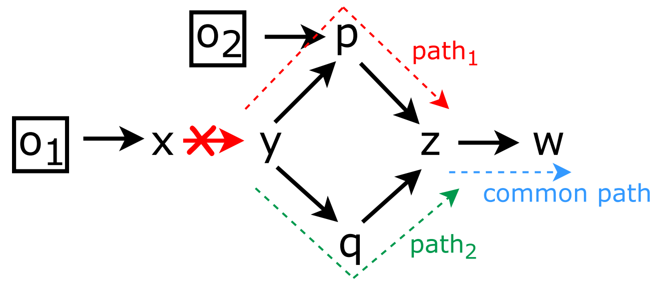

Figure 2.6: An example of parallel change propagation.

- 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 5 and line 7 in Algorithm 6) is synchronized, because different parallel tasks may concurrently add different new edges to the worklist.

2.6.1 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 5 or Algorithm 6, 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 5, 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 5, 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 6, 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 6, 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 $pts(y)$.

2.7 End-to-end Incremental Points-to Analysis

In this section, we present an end-to-end incremental pointer analysis for real-world Java programs based on our new incremental algorithms described in Section 2.5. The presented pointer analysis is context-, path- and flow-insensitive.

Algorithm 7: Incremental Pointer Analysis for Java

```
  Input : Δ_{P_IR} - a set of IR changes.
          Deletions: D: -{d1,d2,...};
          insertions: I: +{i1,i2,...}.
  // for each deleted IR statement
1 foreach s ∈ D do
      // extract edge(s) according to Table 2.2
2     e ← ExtractEdge(s)
      // call Algorithm 2 for each deleted edge
3     DeleteEdge(e)
4 end
  // for each inserted IR statement
5 foreach s ∈ I do
      // extract edge(s) according to Table 2.2
6     e ← ExtractEdge(s)
      // call Algorithm 4 for each added edge
7     AddEdge(e)
8 end
```

Table 2.2: Java statements and their corresponding PAG edges.

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

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

Consider a programming environment where the developer performs an initial commit of his project, we compile the project, translate Java bytecode to an SSA-based IR [50] and use the IR to construct a PAG. Then, the developer has committed a collection of program changes $\Delta_{P_{source}}$. We recompile the project with $\Delta_{P_{source}}$ 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[2]. $\Delta_{P_{IR}}$ can be divided into two disjoint sets: D - a set of old IR statement deletions and I - a set of new IR statement insertions.

Algorithm 7 shows our end-to-end algorithm. For each IR statement, we first extract the corresponding edges in the PAG according to Table 2.2. 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 allocate site and has a type T.

❷ (**simple assignment**): an edge from a pointer node y to x.

[2] 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.

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 2.1). 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. There are also several subtle statements related to exceptions and reflection, which can be handled by existing techniques [51, 52].

For each identified edge, we then call Algorithm 2 if it is deleted and Algorithm 4 if added, to compute the new points-to information and update the PAG. To parallelize our algorithm, in each iteration of Algorithms 2 and 4, we call Algorithms 5 and 6 to propagate the points-to set changes in parallel. On a multicore machine, we can maintain a thread pool to perform the parallel tasks.

2.7.1 Adapting To Context-sensitive, Flow-sensitive And Other Problems

Heretofore, our incremental and parallel algorithm is field-sensitive and object-sensitive for collection objects. Next, we show the potential to extend our analysis to other dimensions in pointer analysis or even other research problems.

2.7.1.1 Context-sensitive

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 [36, 53, 54] and CFL-reachability [41, 42, 55, 56]. Recursive calls is hard to handle for context-sensitive pointer analysis in both types [57]. Collapsing the calls within a SCC to a single call node in call graph will lead to a precision loss. Instead, to detect a *recursive data structures* in a PAG [55] can maintain higher precision and is also the presumption of our work (*i.e.*, an acyclic PAG).

In a k-CFA pointer analysis, pointer variables uses *k* call strings from the call graph to distinguish contexts. The whole analysis is built on PAG, but it maintains a separate abstract pointer $\langle x, [c_1, c_2, ...c_k]\rangle$ for each local variable x to represent the points-to sets at a call site c_k, given the caller context $[c_1, c_2, ...c_{k-1}]$, and a set of abstract memory allocations $\langle o, [c_1, c_2, ...c_k]\rangle$ to determine what context should be used. When there is an incremental code change, we can follow the Algorithm 2 and 4 to perform the update for each edge with an additional criterion: we only consider the points-to sets with sub call strings from $[c_1, c_2, ...c_k]$ when checking the incoming neighbours and propagating changes to outgoing neighbours.

CFL-reachability pointer analysis also represents a program by a PAG, where 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. Take a trace-based incremental CFL-reachability pointer analysis [42] as example. Initially, we should answer all queries from a program and cache the answers and their traces that record the dependency information while obtain the answers. Then, for each incremental edge change $x \rightarrow y$, rather than considering all the neighbours of y in the PAG, we perform Algorithm 2 and 4 along the cached traces that contain y.

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 of variable on the paths has been cached.

2.7.1.2 Flow-sensitive

Flow-sensitive pointer analysis is always expensive and complicated. To determine a points-to set at a program point, SSA representation and control flow graph are always combined to infer points-to and def-use relations and to avoid unnecessary propagation, which are represented by graphs [58, 59, 60]. Sometimes, the analysis can be cast to a graph reachability problem [61].

When incremental changes has been 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 affected points-to sets, and (3) interprocedural data flow changes. Since the points-to relation and control flow info are coupled in the graphs when considering flow-sensitivity, a simple checking 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 info as IncA, 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 info 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.

2.7.1.3 Other Problems

A good extension is to apply our incremental and parallel analysis on other research problems that requires analyzing and updating information on graphs. As long as the analyzed graph $G = (V, E)$ satisfies the following properties, our analysis can be adapted to work on that problem:

- G is a directed graph;
- SCC collapsing can be applied on G with a tolerable precision loss;
- Each node in V represents a set of elements;
- Each edge in E represents a constraint which propagates the elements;
- The direction on an edge indicates the propagation direction;
- The propagation of elements on G can terminate.

2.7.2 Scheduling Of Changed Statements

Since multiple statement additions and deletions need to handle at one time and sometimes an addition 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 workload.

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 additions 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 numbers, the numbers may represents for different variables in the method. Rather than performing a complex procedure to identify the correlations between each added and deleted statements, we perform a simple schedule as described in Algorithm 7

2.8 Related Works And Comparison

The survey [63] has provided a detailed study in pointer analysis. Pointer analysis has been extensively researched along several dimensions, which affect the trade-off between cost and precision, e.g., context-sensitivity [55, 64, 65, 66, 67, 68], flow-sensitivity [58, 59, 60, 61], path-sensitivity [69], field-sensitivity [55, 70], demand-driven [23, 56, 71], algorithmic complexity analysis [72, 73, 74, 75], as well as incremental pointer analysis [37, 38, 39, 41, 43].

2.8.1 Incremental Algorithms

The incremental pointer analysis algorithms for handling dynamic program changes are inadequate. Most existing algorithms [37, 38, 39, 41] assume a static program call graph. Demand-driven analyses [23, 56, 71] can handle program changes in an intuitive way: add/remove edges in the graphs and re-issue a query to compute the updated points-to set. Existing incremental algorithms [12, 41] based on reset-recompute and graph reachability are difficult to scale and parallel and/or reduce the analysis precision. In particular, they cannot handle code deletion efficiently because pointer analysis is non-distributive. 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.

Saha et al. [37, 40] propose an incremental and demand-driven points-to analysis based

on *DRed* algorithm [76] for tabled evaluation, which uses a *deletion-rederivation* strategy that is 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, *supported 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" process redundantly propagates the marks, since it cannot recognize whether an answer should be removed at the first glance. Instead, our analysis can discover whether a node need to be updated immediately. Furthermore, our incremental analysis is designed for massively parallelization. However, the rederivation from the technique requires computing *derivation length* to determine the order in which marked answer should be recomputed first. Such an order prohibited 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.

Reviser [38] 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 $f(a) \sqcap f(b) = f(a \sqcap b)$, but pointer analysis is a particular problem that

does not satisfy such a property, because the effect of code deletion cannot be modeled by merge operations. Hence, IFDS/IDE does not apply to incremental pointer analysis. On the other hand, for the "local neighbour" properties, we identify and prove that the particular problem of pointer analysis satisfies these properties (which is not shown by previous research) and that we can leverage them to improve the analysis efficiency. Nevertheless, these properties 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.

IncA [43] 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. As introduced in Section 2.3, the scalability to large programs is hard to guaranteed.

Incremental computation has been extensively discussed in the domain of datalog evaluation. There are several pointer analyses formulated using datalog frameworks [77, 78, 79, 80, 81]. However, despite intensive research [76, 82] for optimizing the incremental evaluation of datalog, datalog engines are still inefficient to handle incremental deletion of the pointer analysis facts.

2.8.2 Parallel Algorithms

Putta and Nasre [83] 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. [84] 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 [85], the rules are performed in parallel on non-interfering nodes in the constraint graph. This graph rewriting algorithm has been further implemented on GPU [86] with an efficient graph representation for the constraint graph under the GPU memory model. Nagaraj et al. [87] propose a flow-sensitive pointer analysis which is paralleled based on the graph rewriting rules from [84].

PSEGPT [88] 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. PSEGPT involves four analysis operations that propagate points-to information, and they can be executed in parallel if they obey the data dependence among operations.

Edvinsson et al. [89] 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 invoke.

However, all these 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 and a pre-built call graph. Our parallel analysis is based on the change consistency property upon the incremental pointer analysis, which has not been proposed in other work. Besides, our parallel algorithm shares no key insight with the existing parallel techniques.

Su et al. [90] 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. Such parallelism is hard to design.

2.8.3 SCC Optimizations

A number of pointer analysis algorithm [45, 62, 68, 91] have adopted some SCC optimizations to further boost their performance, *e.g.*, Tarjan's algorithm [48]. However, all these optimizations do not update SCCs according to dynamic graph changes as we do in our incremental algorithm.

La Poutré et al. [46] propose the first algorithm to dynamically maintain the transitive reduction of a directed graph w.r.t. graph edge additions and deletions, i.e., dynamically collapse/break SCCs. Bergmann et al. [47] adapt the algorithm [46] for incremental graph pattern matching to improve the scalability.

Marlowe et al. [92] propose an incremental data flow analysis which decompose/compose the affected SCCs in the data flow graph whenever there are data flow changes, so that a precise and correct result can be obtained efficiently.

Since [46] is the most classic and efficient algorithms to perform the SCC update, we adopt its main idea in our incremental analysis.

3. PARALLEL INCREMENTAL HAPPENS-BEFORE ANALYSIS [1]

Happens-before analysis is necessary during concurrency bug detection, since it represents a sequence between two events, such that if one event should happen before or after another event. Section 3.1 introduces the background of happens-before analysis, Section 3.2 provides the classic techniques in happens-before analysis, Section 3.3 introduces an example to illustrate the limitations in existing techniques (Section 3.4) and our new static happens-before analysis (Section 3.5 and 3.6). Finally, we explain how to discover the happens-before relation between two events in our analysis.

3.1 Background

Happens-before analysis computes the happens-before relation between two events, which is a relation between the result of two events, such as if one event should happen before or after another event, or the two events do not have any relation. The happened-before relation is formally defined as the least strict partial order on events such that:

1. If events a and b occur on the same process, $a \rightarrow b$ if the occurrence of event a preceded the occurrence of event b;
2. If event a is the sending of a message and event b is the reception of the message sent in event a, $a \rightarrow b$.

Like all strict partial orders, the happened-before relation is transitive, irreflexive and antisymmetric. In Java specifically, a happens-before relation is a guarantee that memory write access by statement a is visible to memory read access by statement b, that is, that statement a completes its write before statement b starts its read.

[1]Reprinted with permission from "D4: Fast Concurrency Debugging with Parallel Differential Analysis" by Bozhen Liu, Jeff Huang, 2018. Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, 359-373, Copyright [2018] by Association for Computing Machinery, Inc. Reprinted by permission.

3.2 Related Works

The most famous algorithm of computing happens-before relations in distributed system should be the vector clock from Lamport timestamps [27], which provide a partial ordering of events with a high overhead. To improve the efficiency of the vector clock, [3] records only the *epoch* of the last read to reduce the number of events we need to record. [28] presents an efficient algorithm, iFT, that uses only the epochs of the access histories, which requires $O(1)$ operations to maintain an access history. Except for the techniques used in dynamic race detection, [12] presents a static happens-before analysis to construct a partial order among read/write events.

3.3 An Example

We use an example in Figure 3.1 to illustrate the limitation of existing techniques and our new algorithm. Initially, we do not have the code on line 11 calling method $m2$. Later, the programmer wants to add the call on line 11, and then remove the statement on line 20 in method $m2$.

```
1 main() {                t1:                t2:
2  x = 0;                 9   x = 1;         12  y = x;
3  y = 5;                 10  m1();          13  m1();
4  t1 = new Thread();     11  m2();//add     14  m2();
5  t2 = new Thread();
6  t1.start();            15  void m1(){     18  void m2(){
7  t2.start();            16   x = 3;        19   x = 2;
8 }                       17   print(x);}    20   y = 0;//del}
```

Figure 3.1: An Example for SHB Analysis.

3.4 Limitations In Existing Techniques

ECHO uses a static happens-before (SHB) graph to compute happens-before relation among abstract threads, memory accesses, and synchronizations. The SHB graph is constructed incrementally following the rules in Table 3.1. Among them, statements ❹ (method $call$), ❺ (thread $start$) and ❻ (thread $join$) generate additional edges according to Table 3.2. The SHB graph is represented by sequential traces containing per-thread nodes in the SHB graph following the program order, connected by inter-thread happens-before edges. For race detection, ECHO computes the happens-before relation between nodes from different threads by checking the graph reachability.

Table 3.1: Nodes in the SHB Graph

Statements	Nodes
❶ $x = y.f^{\star}$	$\forall O_c \in pts(y) : read(O_c.f)$
❷ $x.f = y^{\star}$	$\forall O_c \in pts(x) : write(O_c.f)$
❸ $synchronized(x)\{s...\}$	$\forall O_c \in pts(x) : lock(O_c), unlock(O_c)$
❹ $x = o.m(y)$	$\forall O_c \in pts(o) : call(O_c.m)$
❺ $t.start()$	$\forall O_c \in pts(t) : start(O_c)$
❻ $t.join()$	$\forall O_c \in pts(t) : join(O_c)$

* ❶ and ❷ also represents the array read ($x = y[i]$) and write ($x[i] = y$).

Table 3.2: Edges in the SHB Graph

Statements	Edges
❹ $x = o.m(y)$	$\forall O_c \in pts(o) : call(O_c.m) \rightarrow FirstNode(O_c.m)$ $LastNode(O_c.m) \rightarrow NextNode(call)*$
❺ $t.start()$	$\forall O_c \in pts(t) : start(O_c) \rightarrow FirstNode(O_c)$
❻ $t.join()$	$\forall O_c \in pts(t) : LastNode(O_c) \rightarrow join(O_c)$

* NextNode(call) represents the consecutive node of the method call statement.

Figure 3.2 shows the SHB graph constructed following the rules in ECHO. We can see

the traces in t_1 and t_2 have duplicated nodes representing the same statements in method $m1$. Besides, we need to retraverse the statements in method $m2$ to complete the trace of t_1 after adding the code on line 11, and remove two write nodes for the statement removal of line 20. From theses, we indicate the inefficiency part in this SHB graph construction and update:

Large SHB graphs A crucial limitation of ECHO is that for large software it can produce a prohibitively large SHB graph. During the graph construction, when a method is invoked, ECHO goes into the method and creates new nodes for statements inside the method. If a method is invoked multiple times (invoked repeatedly by a thread, occurs in a loop, or by multiple threads), multiple nodes representing the same statement will be created and inserted into the SHB graph.

Expensive graph update Updating the SHB graph with respect to code changes can be very expensive. ECHO uses a map to record each method call and its corresponding location in the SHB graph. If there is a statement change in a method, ECHO has to track and update all the matching nodes in the graph. For large software, this incurs significant repetitive work because a changed method can be invoked many times.

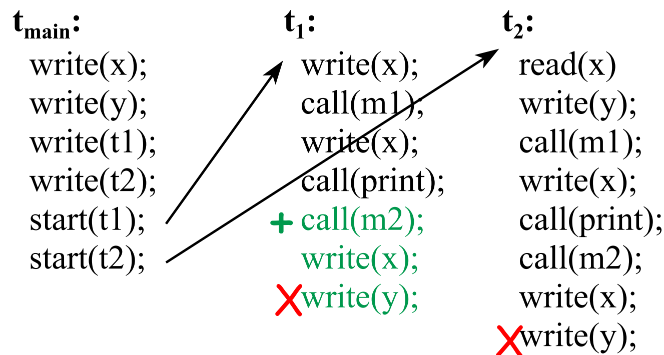

Figure 3.2: The SHB graph for the example in Figure 3.1.

3.5 A New SHB Graph Construction

A key to our scalable happens-before analysis is a new representation of the SHB graph, which enables both compact graph storage and efficient graph updating. Unlike ECHO which constructs per-thread sequential traces with repetitive nodes corresponding to the same statement, we construct a unique subgraph for each method/thread and connect the subgraphs with happens-before edges.

We maintain a map $exist$ from the unique id of each method/thread to its subgraph $subshb_{id}$. Each subgraph has two fields: $tids$ which records the threads which have invoked/forked the method/thread, and $trace$ which stores the SHB nodes corresponding to the statements inside the method/thread. Taking the main method ($target$), an empty subgraph ($subshb_{tar}$) and the executing thread id ($ctid$) as input, the algorithm returns the SHB graph (shb). Initially, we add the pair of <tar, $subshb_{tar}$> to the $exist$ map and include $ctid$ into the field $tids$ of $subshb_{tar}$. Afterwards, we extract the statements in the $target$ and create SHB nodes according to Table 3.1 for each statement and insert it into $subshb_{sig}.trace$.

Table 3.3: Edges in the New SHB Graph

Statements	Edges
❹ $x = o.m(y)$	$\forall O_c \in pts(o) : call(O_c.m) \xrightarrow{t_{id}} subshb_{O_c.m}$
❺ $t.start()$	$\forall O_c \in pts(t) : start(O_c) \xrightarrow{t_{id}} subshb_{O_c}$
❻ $t.join()$	$\forall O_c \in pts(t) : subshb_{O_c} \xrightarrow{t_{id}} join(O_c)$

The new happens-before edges are constructed according to Table 3.3. Each edge is labeled with the thread id. For method call ❹, we create a unique signature sig of each callee method $O_c.m$ and check the map $exist$ if $subshb_{sig}$ has been created. If sig exists, it means $O_c.m$ has been visited before and its subgraph has been created, which avoid

redundant statement traversal. We thus add the $ctid$ into $subshb_{sig}.tids$ and add a new happens-before edge from the calling node to the new subgraph with the label $ctid$. Otherwise, we create a new subgraph $subshb_{sig}$ for the newly discovered method. For thread start ❺, we create a new thread id (tid) for each object node in $pts(t)$, and follow the same procedure to construct $subshb_{tid}$ and add happens-before edges. For thread join ❻, we add an edge from the last node in $subshb_{tid}$ to the join node in $subshb_{tar}$, where tid is the thread id of the joined thread, corresponding to the object node in $pts(t)$. The procedure for creating different subgraphs can run in parallel, since different threads/methods are independent from each other.

We use an example in Figure 3.1 to illustrate our algorithm. Suppose the method call $m2()$ at lines 11 is not in the program initially. We first create $subshb_{main}$ and traverse the statements in main method. After inserting $write(x)$ and $write(y)$ into the $trace$ field for the two writes at lines 2 and 3, we see the two thread start operations. We then create $subshb_{t_1}$ and $subshb_{t_2}$ for the two threads in parallel and add their corresponding happens-before edges. Consider the two method calls $m1()$ at lines 10 and 13, they introduce only one subgraph $subshb_{m1}$, which is created when $m1()$ is visited the first time. The final SHB graph is shown in Figure 3.3.

3.6 The New SHB Graph Update For Incremental Changes

Thanks to our new SHB graph representation, incremental changes can be updated efficiently in parallel: 1) changes to statements in a method that is invoked multiple times need to be updated only once (instead of multiple times in ECHO); and 2) multiple changes to different methods/threads can be updated in parallel (because they belong to different subgraphs).

For each added statement, we simply follow the same SHB graph construction procedure described in the previous subsection. For each deleted statement s, we first delete

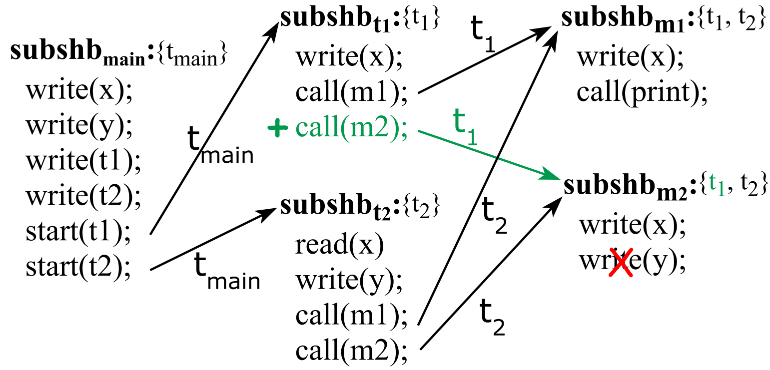

Figure 3.3: The new SHB graph for the example in Figure 3.1.

the node representing s from its belonging $subshb_{tar}$. In addition, for method call ❹, we locate the subgraph of the callee method and remove the corresponding SHB edges. For thread start ❺, we remove the corresponding SHB edges for each $subshb_{tid}$. Note that we do not remove the subgraph itself, such that the subgraph can be reused later if the method call or thread start is added back. For thread join ❻, we remove the SHB edge from $subshb_{tid}$ to $subshb_{tar}$.

Consider two changes in our example in Figure 3.1: (i) inserting a method call statement $m2()$ at line 11, and (ii) deleting the statement at line 20. For (i), we first create a method call node $call(m2)$ at the last position in $subshb_{t_1}$. Since $subshb_{m_2}$ already exists in the SHB graph, we skip traversing $m2()$ and then add an edge $call(m2) \xrightarrow{t_1} subshb_{m_2}$ to the graph. For (ii), we localize the $write(y)$ node corresponding to this statement and simply remove it from $subshb_{m_1}$.

3.7 How To Compute The Happens-before Relation

Our new SHB graph representation also makes computing the HB relation more efficient compared to ECHO. Different from ECHO which checks path reachability between each individual pair of nodes, for changes in a method invoked multiple times, we can

check for multiple node pairs altogether. For example, in Figure 3.3, although the method $m2()$ is invoked twice by t_1 and t_2 which generates two write nodes, when computing the HB relation between the nodes in t_{main} and those from $m2()$, we can find that the nodes in t_{main} dominate all nodes in $m2()$ in the SHB graph. Therefore, we can determine the happens-before relation for all these two write nodes by checking the path dominator once.

4. D4: A FAST CONCURRENCY DEBUGGING FRAMEWORK [1]

Concurrency bug detection always refers to data race detection, deadlock detection, atomicity violation, which has been extensively developed during the last two decades. In this thesis, we focus on data race and deadlock detection, but our analysis can be extended to discover atomicity violation easily. This chapter introduces the background of data race and deadlock detection (Section 4.1), the related work in this area (Section 4.2), and most importantly, our fast concurrency analysis framework: D4 [30]. Specifically, we introduce how to extract minimal changes from the previous updated graphs to guarantee an efficient incremental detection (Section 4.3), the data race and deadlock detection insides our framework (Section 4.4 and 4.5), and the detail of the distributed framework (Section 4.6).

4.1 Background

Data races and deadlocks are the most notorious concurrency bugs in multi-threaded programs. Firstly, we introduce the concepts of data race and deadlock based on our thesis.

A data race occurs when it satisfies the following conditions:

1. two or more threads access the same memory location concurrently, and
2. at least one of the accesses performs write access
3. there is no exclusive locks used to order their accesses to that memory

When these three conditions hold, the order of accesses is non-deterministic, and the computation can give different results from run to run depending on the access order.

[1] Reprinted with permission from "D4: Fast Concurrency Debugging with Parallel Differential Analysis" by Bozhen Liu, Jeff Huang, 2018. Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, 359-373, Copyright [2018] by Association for Computing Machinery, Inc. Reprinted by permission.

Deadlocks usually occur when the order of locks is not consistent in the program. For example, *thread1* acquires *lock1* and is going to acquire *lock2*, while *thread2* acquires *lock2* and is going to acquire *lock1*. In this case, deadlock happens because threads are waiting for each other to release the second lock and the program will hang there.

4.2 Related Works

This research area has been extensively developed, and there are a lot of techniques to refer to, such as [1, 2, 4, 6, 8, 10, 11, 93, 94, 95] detect concurrency bugs for late phases of software development, e.g., testing or production, when the whole program is completed. Hence, it is often hard to scale these techniques, too difficult to understand a detected bug, or even too late to fix a bug.

4.3 Change Extraction

As listed in Table 3.1, there are six types of statements that can lead to differences in points-to sets, data race and deadlock detections. We present the following three changes and the influences caused by the changes:

1. points-to sets: statement ❶ and ❷ can affect the points-to sets of a variable, which thus may lead to the overlap of the points-to sets of two variables and changes the guarded lockset of a variable;

2. lock dependency: statements ❸ lead to the addition or deletion of lock pairs and change the guarded lockset for a variable, which can introduce a potential deadlock;

3. happens-before relation: statement ❹ to ❻ influence the happens before relation between statements and therefore it may expose/hide the data races/deadlocks;

All the above changes can produce or eliminate data races and deadlocks in a program. Algorithm 8 illustrate how we extract the changes in the above updated graph to guarantee the efficiency and soundness of concurrency bug detection after an incremental code

change. First, it collects a set of nodes $\mathcal{N}$ containing all the `read` and `write` nodes that require to be re-evaluated. Second, it considers all the nodes that may happen-in-parallel with $\mathcal{N}$ to identify if a race is introduced. The algorithm takes the statement changes $stmts$, the updated SHB graph shb and the output $changes$ as input, where $changes$ contain the variables of which the points-to sets have been changed.

To compute $\mathcal{N}$, for each variable v in $changes$, the algorithm first adds all the read and write nodes related to v from the SHB graph to $\mathcal{N}$ (*line 2 and 3*). Afterwards, for each changed statement s in $stmts$, the algorithm considers two cases. If s is a lock operation, the algorithm first locates the influenced $lock - unlock$ pair in the SHB graph and adds all the read and write nodes between them to $\mathcal{N}$. Otherwise, if s is a method call a thread start or join operation, the algorithm first locates the $subshb$ for the method or thread and then adds all the read and write nodes in $trace$ to $\mathcal{N}$ (*line 5-13*).

Algorithm 8: The overview of change extraction

Input : $stmts$ - incremental statement changes
$changes$ - incremental PAG changes
shb - updated SHB graph

Output: $\mathcal{N}$ - the read/write nodes need to be re-evaluate

```
1  N ← ∅
2  foreach v ∈ changes do
3  |   N ← ExtractNodesOf(v)
4  end
5  foreach s ∈ stmts do
6  |   if s instanceof ② or ③ then
7  |   |   ⟨lock, unlock⟩ ← LocateNodes(s)
8  |   |   N ← ExtractNodesBtw(⟨lock, unlock⟩)
9  |   end
10 |   if s instanceof ④ or ⑤ then
11 |   |   trace_i ← LocateTrace(s)
12 |   |   N ← ExtractNodesIn(trace_i)
13 |   end
14 end
```

4.4 Data Race Detection

In this section, we briefly introduce the data race detection in our framework. Algorithm 9 presents how our system detects a data race related to the changes to the program. Taking the output $\mathcal{N}$ from Algorithm 8 as input, we distribute each read/write node to worker thread to let them check if there exist conflict memory accesses with this node. If yes, the worker thread reports the race warning to the master scheduler. Finally, we report all the detected races to users.

Performing a static race detection for whole programs can follow the same algorithm, but the input $\mathcal{N}$ should be all the read/write nodes in the initial SHB graph.

Algorithm 9: The overview of data race detection

Input : $\mathcal{N}$ - the read/write nodes need to be re-evaluate
shb - updated SHB graph
ld - updated LD graph
Output: $races$ - detected data races

```
   // parallel work distribution in master
1  ParallelDetectionMaster(N):
2  while N ≠ ∅ do
3  |   n ← N.removefirst()
4  |   TellWorker(n)
5  |   foreach msg = ⟨n1, n2⟩ do
6  |   |   races ← msg
7  |   end
8  end

   // data race detection in worker
9  ParallelDetectionWorker(n):
10 mhps ← CollectMHPNodesFor(n)
11 foreach v ∈ mhps do
12 |   if PerformDataRaceDetection(n, v) then
13 |   |   TellMaster(⟨n, v⟩)
14 |   end
15 end
```

4.5 Deadlock Detection

In this section, we focus on our novel incremental deadlock detection algorithm atop D4. Although exhaustive algorithms for deadlock detection exist, this is the first incremental deadlock detection algorithm, which is in fact highly non-trivial without D4. One has to develop new incremental data structures, update them correctly upon code changes, and integrate them efficiently with incremental race detection. Next, we first introduce the *lock-dependency graph* which can be constructed from the SHB graph. Then we present our incremental algorithm that uses the graph for deadlock detection.

4.5.1 Lock-dependency Graph

The lock-dependency (LD) graph contains nodes corresponding to lock operations, and edges corresponding to lock dependencies. For example, if a thread t is holding a lock l_1 and continues to acquire another lock l_j, an edge $lock(l_1) \xrightarrow{t} lock(l_2)$ is added to the LD graph.

The LD graph can be constructed from the SHB graph by traversing the lock/unlock nodes for each thread. For a lock statement on variable p, suppose $pts(p)$ = $\{o_1,o_2\}$, it generates two lock nodes in the LD graph: $lock(o_1)$ and $lock(o_2)$. Figure 4.1 shows an example. The LD graph contains three nodes $lock(o_1)$, $lock(o_2)$ and $lock(o_3)$ connected by edges labeled with corresponding thread ids.

4.5.2 Deadlock Detection

Our basic idea of deadlock detection is also to look for circles in the LD graph with edge labels from multiple threads, which indicates circular dependencies of locks. We then check the happens-before relation between the involved nodes to find real deadlocks. For example, in Figure 4.1(b), $lock(o_1) \xrightarrow{t_1} lock(o_2)$ and $lock(o_2) \xrightarrow{t_2} lock(o_1)$ forms a circular dependency. To realize incremental deadlock detection, we develop an incremental

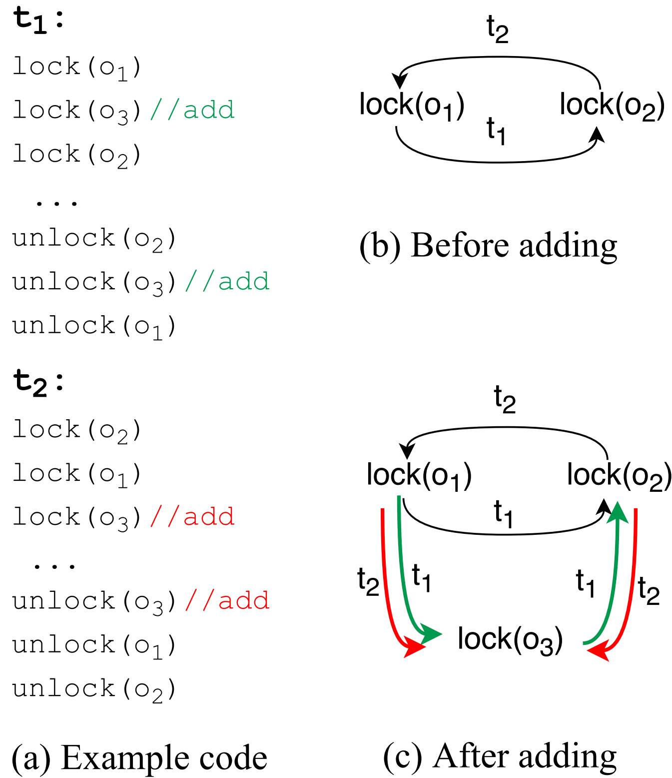

Figure 4.1: An example for the LD graph construction.

algorithm for updating the LD graph and an incremental algorithm for deadlock checking.

4.5.3 LD Graph Update For Incremental Changes

For an added synchronized statement in thread t, we first locate its belonging method and its corresponding subgraph $subshb_{tar}$, and create a pair of $lock/unlock$ nodes and insert them into $subshb_{tar}$ according to the statement location. Starting from the changed $node$, we search the first $lock/unlock$ node right before the added $lock$ node ($pred$), and the consecutive $lock/unlock$ node right after the added $lock$ node ($succ$) along edges in the SHB graph. We call two $lock$ nodes connected by an edge a *lock pair*. If $pred$ is a $lock$ node, it means $pred$ and $node$ can form a lock pair with thread ids in $subshb_{tar}.tids$. Meanwhile, if $succ$ is also a $lock$ node, a lock pair between $node$ and $succ$ is added to

the LD graph. Afterwards, we reversely traverse the LD graph to discover the incoming $lock$ nodes of $pred$ with edges labeled t ($pred'$). For each $pred'$, we add a new lock pair between $pred'$ and $node$. Then, we collect the outgoing $lock$ nodes of $succ$, and create lock pairs for $node$ and each of them. For a deleted synchronized statement, we simply remove its corresponding $lock/unlock$ nodes from $subshb_{tar}$ as well as its lock pairs.

Consider Figure 4.1(a) in which $lock(o_3)$/$unlock(o_3)$ are added in both t_1 and t_2. We first localize the lock nodes before and after the added statement, and then add four edges: $lock(o_1) \xrightarrow{t1} lock(o_3)$, $lock(o_1) \xrightarrow{t2} lock(o_3)$, $lock(o_3) \xrightarrow{t1} lock(o_2)$ and $lock(o_2) \xrightarrow{t2} lock(o_3)$, as shown in Figure 4.1(c).

4.5.4 Incremental Deadlock Detection

Algorithm 10 illustrates the incremental deadlock detection. The key idea is to check only the circles containing the changed (added or deleted) $lock$ nodes. We first collect all the circular dependencies that include the changed $lock$ nodes. Then, we parallel the deadlock detection for all circles by checking the happens-before relation between conflicting lock and unlock nodes from different threads in each circle.

4.6 The Distributed System Design

In this section, we present the design of our distributed analysis framework, D4. There are three main components in our design of distributing the analysis to a remote server, which is expected to have much more computation power than the machine running the IDE. The first component is a change tracker that tracks the code changes in the IDE and sends them to the server with a compact data format. The second component is a real-time parallel analysis framework that implements our incremental algorithms for pointer analysis and happens-before analysis. The third component is an incremental bug detector that leverages our framework to detect concurrency bugs and send them back to the IDE. We next focus on describing the second component, which is the core of our system.

Algorithm 10: IncrementalDeadlockDetection

Global States: shb - updated SHB graph
ldg - updated LD graph
Input : Δ_{lock} - the changed lock nodes
Output : $deadlocks$ - detected deadlocks

```
1  circles ← DiscoverCircularDependency(ldg, Δ_lock)
2  foreach c ∈ circles do
3  │ ParallelDeadlockDetection(c)
4  end

5  ParallelDeadlockDetection(c):
6  tids ← ExtractTidsInCircle(c)
7  foreach (t_i, t_j) ∈ tids do                    // for each pair of threads
8  │ lock(x), lock(y) ← FindConflictingLocks(t_i, t_j, c)
   │ // check happens-before condition
9  │ if (!CheckHBFor(lock(x)_{t_i}, lock(y)_{t_j}) && !CheckHBFor(lock(x)_{t_j}, lock(y)_{t_i}))
   │  then
10 │ │ deadlocks ← c
11 │ end
12 end
```

4.6.1 Parallel Analysis Framework

We implement a communication interface between the client and the server based on the open-source Akka framework [96], which supports efficient real-time computation on graphs via message passing and asynchronous communication. Akka is based on the actor model and distributes computations to actors in a hierarchical way. We hence can run the server on both a single multicore machine or multiple machines with a master-workers hierarchy. The master actor manages task generation and distribution, and the worker actor performs specific graph computations (e.g., adding/removing nodes/edges and updating the points-to sets). Tasks are assigned by the master and consumed by workers following a work stealing schedule until there is no more remaining task.

4.6.2 Graph Storage

Due to the distributed design, we can leverage distributed memory to store large graphs when the memory of a single computing node is limited. For the PAG, we partition the graph by following the edge cut strategy in Titan [97], in which nodes/edges created from the same method and those involved in the same points-to constraint are more likely to be stored together. For the SHB graph, we separate it into two parts: graph skeleton and subgraphs. The graph skeleton uses SHB edges to connect the ids of subgraphs and can be stored in a single memory region. The subgraphs can be stored in different memory regions and located efficiently by maintaining a map between each subgraph and its id.

4.6.3 Message Format

Akka provides protocol buffers and custom serializers to encode messages between client and server. We encode all graph nodes/edges and subgraph ids as integers or strings to facilitate message serialization. For example, deleting a statement "`b=a`" is encoded as "`-id`" where `id` is the unique id of the statement in the SSA form, and it is further encoded into "`-(id1,id2)`" on the server for graph computation, in which `id1` and `id2` represent integer identifiers of nodes `a` and `b` respectively, and `id1` is the source and `id2` the sink of the PAG edge.

5. EVALUATION [1]

We implemented D4 based on ECHO and evaluated it on a collection of 13 real-world large Java applications from DaCapo-9.12 [98], as shown in Table 5.1. We ran the D4 client on a MacBook Pro laptop with Intel i7 CPU and the server on a Mercury AH-GPU424 HPC server with Dual 12-core Intel© Xeon© CPU E5-2695 v2@2.40GHz (2 threads per core) processors. In this section, we report the results of our experiments.

5.1 Evaluation Methodology

For each benchmark, we run three sets of experiments. (1) We first run the whole program exhaustive analysis on the local client machine to detect both data-races and deadlocks. Then, we initialize D4 with the graph data 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 D4, which uses the parallel incremental algorithms for detecting concurrency bugs. (3) For the deleted statement in the previous step, we add it back and re-run D4.

We run D4 with two server configurations: a single thread (*D4-1*) and 48 threads (*D4-48*). We measure the time taken by each component in each step and compare the performance between the exhaustive analysis and D4. In addition, we repeat the same experiments for ECHO running on the client machine to compare the performance of D4 with ECHO.

[1]Reprinted with permission from "D4: Fast Concurrency Debugging with Parallel Differential Analysis" by Bozhen Liu, Jeff Huang, 2018. Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, 359-373, Copyright [2018] by Association for Computing Machinery, Inc. Reprinted by permission.

5.2 Benchmarks

The metrics of the benchmarks and their PAGs are reported in Table 5.1. Columns 2-6 report the numbers of classes, methods, pointer nodes, object nodes and edges in the PAG of each benchmark, respectively. More than half of the benchmarks contain over 1M pointer nodes and over 200M edges in the PAG. The default pointer analysis is based on the ZeroOneContainerCFA in WALA, which 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 to ensure that the exhaustive analysis can finish within 6 hours.

Table 5.1: Benchmarks and PAG metrics.

App	#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

5.3 Evaluation Of PIPTA

Table 5.2 and 5.3 compares the performance between exhaustive pointer analysis and different incremental algorithms.

Overall, D4 achieves dramatic speedup over the other algorithms, especially for han-

dling deletion. For most benchmarks, the exhaustive analysis takes several hours to compute (2.4h on average). For a deletion change, on average, the *reset-recompute* incremental algorithm in ECHO takes 26s, the *reachability*-based incremental algorithm in ECHO takes 39s, whereas *D4-1* and *D4-48* take only 73ms and 24ms respectively to analyze, which is three orders of magnitude faster. 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, while *D4-1* and *D4-48* take only 1.1mins and 5.5s respectively for all benchmarks on average, which achieves 20X-200X speedup.

For insertion changes, the average time for all the four incremental algorithms per change is within 0.1s, indicating that these algorithms are fast enough for practical use in the programming phase with respect to incremental code insertions (but not deletion). Nevertheless, for *reset-recompute* and *reachability* the worst case scenarios still take over 7s on average, which could be intrusive in the IDE. However, *D4-1* improves the performance to 4.1s, and *D4-48* further reduces the time to 0.6s, which is reasonably fast for practical use.

5.4 Evaluation Of D4

We will discuss the evaluation of D4 from the following two perspectives: performance and precision.

5.4.1 Performance

Table 5.4 reports the performance of concurrency bug detection for all the seven multi-thread applications in DaCapo, including the time taken by exhaustive analysis, by ECHO (for race detection only), and by *D4-1* and *D4-48* (for both data race and deadlock detection). Note that the time for exhaustive analysis includes constructing both the PAG and the SHB graph for the whole code base and detecting both data races and deadlocks in the

Table 5.2: Performance of Exhaustive and Existing Incremental Pointer Analysis Algorithms.

App	Exha-ustive	ECHO-Reset-Recompute				ECHO-Reachability			
		insert		delete		insert		delete	
		avg.	worst	avg.	worst	avg.	worst	avg.	worst
avrora	4.79h	27ms	3s	52s	>0.5h	32ms	3s	76s	>0.5h
batik	4.03h	6ms	2s	48s	22min	6ms	2.2s	79s	>0.5h
eclipse	1h	5ms	1s	14s	12min	7ms	1.1s	20s	15min
fop	3.3h	12ms	7s	31s	16min	11ms	7.2s	38s	21min
h2	3.9h	11ms	21s	37s	25min	12ms	19s	82s	>0.5h
jython	3.2h	4.2ms	21s	43s	17min	4.5ms	20s	67s	>0.5h
luindex	2.9h	5.2ms	11s	22s	10min	5.3ms	12s	31s	12min
lusearch	2.5h	2.2ms	1.2s	17s	7min	2.4ms	1s	11s	8min
pmd	39min	1.8ms	0.7s	14s	>0.5h	1.8ms	0.7s	14s	>0.5h
sunflow	3.5h	2.8ms	15s	47s	11min	2.2ms	16s	61s	18min
tomcat	35min	9ms	8.7s	9.8s	>0.5h	8ms	9s	12s	>0.5h
tradebeans	45min	1ms	0.6s	3.5s	7min	0.9ms	0.6s	3s	9min
tradesoap	49min	1ms	0.7s	4s	10min	1ms	0.6s	5s	11min
Average	2.4h	6.8ms	7.2s	26s	>17min	7.2ms	7.5s	39s	>22min

Table 5.3: Performance of Exhaustive and New Incremental Pointer Analysis Algorithms.

App	Exha-ustive	D4-1				D4-48			
		insert		delete		insert		delete	
		avg.	worst	avg.	worst	avg.	worst	avg.	worst
avrora	4.79h	0.99ms	1s	89ms	3.8min	0.82ms	0.1s	27ms	10.5s
batik	4.03h	0.86ms	0.8s	95ms	51s	0.41ms	0.1s	42ms	6.1s
eclipse	1h	0.74ms	0.4s	65ms	21s	0.62ms	0.07s	23ms	2.6s
fop	3.3h	1.33ms	5s	110ms	2.9min	0.82ms	0.3s	29ms	5.9s
h2	3.9h	0.18ms	17s	78ms	2min	0.16ms	1.1s	24ms	9.4s
jython	3.2h	0.49ms	12s	96ms	6min	0.35ms	0.9s	18ms	22s
luindex	2.9h	1.1ms	9s	143ms	2.7min	0.88ms	1.7s	31ms	7s
lusearch	2.5h	0.83ms	0.6s	15ms	44s	0.42ms	0.2s	9ms	2.8s
pmd	39min	0.61ms	0.2s	67ms	27s	0.53ms	0.1s	13ms	1.2s
sunflow	3.5h	0.87ms	7s	66ms	1.5min	0.66ms	2.9s	36ms	8s
tomcat	35min	0.32ms	0.3s	64ms	19s	0.19ms	0.05s	28ms	1.8s
tradebeans	45min	0.45ms	0.3s	24ms	14s	0.37ms	0.1s	11ms	0.8s
tradesoap	49min	0.62ms	0.3s	31ms	18s	0.43ms	0.2s	15ms	1s
Average	2.4h	0.72ms	4.1s	73ms	1.1min	0.51ms	0.6s	24ms	5.5s

whole program. The time for ECHO and D4 includes that taken by incremental algorithms for updating the graphs and detecting bugs per change.

Overall, the exhaustive analysis requires a long time (>2h on average) to detect races and deadlocks in the whole program. The incremental detection algorithms are typically orders of magnitude faster than the exhaustive analysis, even in the worst case scenarios. Between D4 and ECHO, the incremental race detection algorithm implemented on top of D4 is much faster than ECHO, achieving 20X-1000X speedup for all cases on average, and 5X-50X speedup for the worst cases. ECHO takes 1.2min on average and 36min in the worst case to detect data races upon a change, while *D4-1* and *D4-48* take only 3.4s and 54ms respectively on average, and 4.9min and 39s in the worst case.

The incremental deadlock detection in D4 is also very efficient. It takes less than 0.2s on average and 36s in the worst case for *D4-1*, and 9ms and 7.7s for *D4-48* per change. Compared to exhaustive analysis, it is over 1000X faster.

Table 5.4: Performance of Concurrency Bug Detection.

App	Exha-ustive	Race Detection						Deadlock Detection			
		ECHO		D4-1		D4-48		D4-1		D4-48	
		avg.	worst	avg.	worst	avg.	worst	avg.	worst	avg.	worst
avrora	>6h	3min	1.8h	21s	15min	231ms	2min	16ms	2min	23ms	32s
batik	>6h	5.2min	2h	1.3s	13min	<1ms	11ms	0.9s	57s	<1ms	8ms
eclipse	1.2h	5s	10min	0.3s	5min	110ms	2min	152ms	49s	13ms	4s
h2	4h	1.2s	6min	33ms	39s	<1ms	18ms	12ms	15s	<1ms	10ms
jython	3.3h	1s	5min	19ms	20s	0.43ms	242ms	17ms	11s	<1ms	53ms
luindex	3h	43ms	2min	4ms	7s	32ms	29s	1.9ms	3.8ms	25ms	17s
lusearch	2.6h	19ms	1.7min	7ms	5s	<1ms	3ms	2.2ms	4.1ms	<1ms	1.3s
Average	>2h	1.2min	36min	3.4s	4.9min	<54ms	39s	0.16s	36s	9.3ms	7.7s

5.4.2 Precision

Although D4 focuses on improving scalability and efficiency through incremental analysis, it does not sacrifice precision compared to the exhaustive analysis. Being a static analysis (which is generally undecidable), D4 can report false positives, but it achieves the same precision as any whole-program static analyzers running the same bug detection algorithm. On the other hand, the warnings reported by D4 are more manageable, because they are reported continuously driven by the current code changes, instead of providing the user with a long list of warnings by analyzing the whole program once.

Table 5.5 shows the results of data races and deadlocks detected by D4 in these benchmarks. Each race or deadlock has a unique signature (no duplicate locations). In total, D4 reports a large number of data races and deadlocks, though many of them may be false positives. We manually inspected 50 randomly selected warnings for each benchmark by checking the source code and the publicly available bug databases. Column "*true*" reports the number of true data races or deadlocks we confirmed (together with the number of known data races or deadlocks which are recorded in the bug databases). Column "*false*" reports the number of false positives. Column "*?*" reports the number of warnings which are uncertain (because we cannot confirm). Overall, more than half of them are real bugs. For example, for *Eclipse*, D4 reports 85 data race warnings and 6312 deadlock warnings. Among the 25 races we inspected, 18 are real (of which 13 are known in the bug database), and 7 are false positives. Among the 25 deadlocks we inspected, 13 are real (of which 10 are known in the bug database), 7 are false positives and 5 are uncertain. For the false positives, we found that a majority of them are related to array accesses, for which we plan to improve precision in future work.

Table 5.5: Results of Detected Concurrency Bugs.

App	#Data Race				#Deadlock			
	total	true	false	?	total	true	false	?
avrora	26419	7(0)	9	12	22	6(0)	8	8
batik	33044	18(10)	5	2	0	-	-	-
eclipse	85	18(13)	7	0	6312	13(10)	7	5
luindex	8776	20(7)	5	0	7249	22(10)	3	0
lusearch	8547	29(29)	12	9	0	-	-	-
h2	0	-	-	-	0	-	-	-
jython	0	-	-	-	0	-	-	-

6. CONCLUSION

Concurrency bud detection (i.e. data race and deadlock detection) is difficult to scale, especially on real-world, large applications. Because the huge amount of thread interleaving and memory accesses requires a long time to analyze. Most of the detection techniques are executed at the late phases of the application design, such as production phase. This makes the expensive fundamental analyses required in the concurrency bug detection to be performed on the whole code bases, which limits the debugging and programming efficiency.

This thesis presents an instantaneous framework for concurrency bug detection, D4, which can indicate the root cause of data races and deadlocks within 0.1s on average according to incremental program changes. This new framework design overcomes the scalability limitation of our prior work, ECHO, which extends ECHO with a client-server architecture.

To support such an efficient framework, we present a new incremental algorithm for pointer analysis that leverages local neighbouring properties for efficient incremental points-to analysis. Compared to existing techniques, our new algorithms achieve orders of magnitude speedup. Moreover, we develop a parallel algorithm that runs efficiently on multicore machines, without redundant recomputation or expensive graph reachability check. The new incremental points-to analysis has 300X-500X speedup comparing with reset-recompute and reachability-based techniques, while our parallel incremental points-to analysis has 1000X-1600X speedup.

Besides, we present a novel incremental algorithm for happens-before analysis that leverages a new representation of the SHB graph for efficient happens-before analysis. Our new representation significantly reduces redundant computations caused by repeated

identical method calls. This boost our concurrency bug detection to have 1.4X-3X speedup on average (not including the speedup from points-to analysis).

Additionally, we provide a distributed system design to utilize the parallelization in a multi-core server, in order to further boost the speed of parallel incremental points-to analysis and happens-before analysis.

In summary, this thesis has presented an instantaneous framework for concurrency bug detection, D4, and two new parallel incremental algorithms in points-to analysis and happens before analysis as the keystone of D4.

REFERENCES

[1] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, "A randomized scheduler with probabilistic guarantees of finding bugs," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, (New York, NY, USA), pp. 167–178, ACM, 2010.

[2] D. Engler and K. Ashcraft, "Racerx: Effective, static detection of race conditions and deadlocks," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, (New York, NY, USA), pp. 237–252, ACM, 2003.

[3] C. Flanagan and S. N. Freund, "Fasttrack: Efficient and precise dynamic race detection," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, (New York, NY, USA), pp. 121–133, ACM, 2009.

[4] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu, "Automated concurrency-bug fixing," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, (Berkeley, CA, USA), pp. 221–236, USENIX Association, 2012.

[5] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, (Berkeley, CA, USA), pp. 267–280, USENIX Association, 2008.

[6] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for java," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, (New York, NY, USA), pp. 308–319, ACM, 2006.

[7] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, pp. 391–411, Nov. 1997.

[8] K. Sen, "Race directed random testing of concurrent programs," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, (New York, NY, USA), pp. 11–21, ACM, 2008.

[9] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: Data race detection in practice," in *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, (New York, NY, USA), pp. 62–71, ACM, 2009.

[10] J. W. Voung, R. Jhala, and S. Lerner, "Relay: Static race detection on millions of lines of code," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, (New York, NY, USA), pp. 205–214, ACM, 2007.

[11] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, "Maple: A coverage-driven testing tool for multithreaded programs," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, (New York, NY, USA), pp. 485–502, ACM, 2012.

[12] S. Zhan and J. Huang, "Echo: Instantaneous in situ race detection in the ide," in *Proceedings of the ? International Symposium on the Foundations of Software Engineering*, FSE âĂŹ16, 2016.

[13] L. O. Andersen, "Program analysis and specialization for the c programming language," tech. rep., 1994.

[14] J. Dietrich, N. Hollingum, and B. Scholz, "Giga-scale exhaustive points-to analysis for java in under a minute," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, (New York, NY, USA), pp. 535–551, ACM, 2015.

[15] D. Grove and C. Chambers, "A framework for call graph construction algorithms," *ACM Trans. Program. Lang. Syst.*, vol. 23, pp. 685–746, Nov. 2001.

[16] B. Hardekopf and C. Lin, "The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, (New York, NY, USA), pp. 290–299, ACM, 2007.

[17] G. Kastrinis and Y. Smaragdakis, "Hybrid context-sensitivity for points-to analysis," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, (New York, NY, USA), pp. 423–434, ACM, 2013.

[18] J.-s. Yur, B. G. Ryder, and W. A. Landi, "An incremental flow- and context-sensitive pointer aliasing analysis," in *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, (New York, NY, USA), pp. 442–451, ACM, 1999.

[19] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to analysis for java," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, pp. 1–41, Jan. 2005.

[20] B. G. Ryder, "Dimensions of precision in reference analysis of object-oriented programming languages," in *Proceedings of the 12th International Conference on Compiler Construction*, CC'03, (Berlin, Heidelberg), pp. 126–137, Springer-Verlag, 2003.

[21] Y. Smaragdakis and G. Balatsouras, "Pointer analysis," *Found. Trends Program. Lang.*, vol. 2, pp. 1–69, Apr. 2015.

[22] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras, "Introspective analysis: Context-sensitivity, across the board," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, (New York, NY, USA), pp. 485–495, ACM, 2014.

[23] J. Späth, L. N. Q. Do, K. Ali, and E. Bodden, "Boomerang: Demand-driven flow- and context-sensitive pointer analysis for java," in *30th European Conference on Object-Oriented Programming (ECOOP 2016)* (S. Krishnamurthi and B. S. Lerner, eds.), vol. 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 22:1–22:26, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016.

[24] M. Sridharan and R. Bodík, "Refinement-based context-sensitive points-to analysis for java," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, (New York, NY, USA), pp. 387–400, ACM, 2006.

[25] M. Sridharan and S. J. Fink, "The complexity of andersen's analysis in practice," in *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, (Berlin, Heidelberg), pp. 205–221, Springer-Verlag, 2009.

[26] J. Whaley and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, (New York, NY, USA), pp. 131–144, ACM, 2004.

[27] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," vol. 21, (New York, NY, USA), pp. 558–565, ACM, July 1978.

[28] O.-K. Ha and Y.-K. Jun, "An efficient algorithm for on-the-fly data race detection using an epoch-based technique," vol. 2015, (New York, NY, United States), pp. 13:13–13:13, Hindawi Publishing Corp., Jan. 2015.

[29] "WALA." `http://wala.sourceforge.net/wiki/index.php/Main_Page`.

[30] B. Liu and J. Huang, "D4: fast concurrency debugging with parallel differential analysis," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pp. 359–373, 2018.

[31] J.-s. Yur, B. G. Ryder, and W. A. Landi, "An incremental flow- and context-sensitive pointer aliasing analysis," in *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, (New York, NY, USA), pp. 442–451, ACM, 1999.

[32] G. Ramalingam, "The undecidability of aliasing," *ACM Trans. Program. Lang. Syst.*, vol. 16, pp. 1467–1471, Sept. 1994.

[33] T. Tan, Y. Li, and J. Xue, "Efficient and precise points-to analysis: Modeling the heap by merging equivalent automata," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, (New York, NY, USA), pp. 278–291, ACM, 2017.

[34] A. S. Robert Smith and J. Gibson, "Poplog's two-level virtual machine support for interactive languages," in *Proceedings of Research Directions in Cognitive Science Volume 5: Artificial Intelligence*, pp. 203–231, 1992.

[35] O. Lhoták and L. Hendren, "Scaling java points-to analysis using spark," in *Proceedings of the 12th International Conference on Compiler Construction*, CC'03, (Berlin, Heidelberg), pp. 153–169, Springer-Verlag, 2003.

[36] P. A. in WALA. `http://wala.sourceforge.net/wiki/\index.php/UserGuide:PointerAnalysis`, 2017.

[37] D. Saha and C. R. Ramakrishnan, "Incremental and demand-driven points-to analysis using logic programming," in *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '05, (New York, NY, USA), pp. 117–128, ACM, 2005.

[38] S. Arzt and E. Bodden, "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*, ICSE 2014, (New York, NY, USA), pp. 288–298, ACM, 2014.

[39] G. Kastrinis and Y. Smaragdakis, "Efficient and effective handling of exceptions in java points-to analysis," in *Proceedings of the 22Nd International Conference on Compiler Construction*, CC'13, (Berlin, Heidelberg), pp. 41–60, Springer-Verlag, 2013.

[40] D. Saha and C. R. Ramakrishnan, "Symbolic support graph: A space efficient data structure for incremental tabled evaluation," in *Logic Programming* (M. Gabbrielli and G. Gupta, eds.), (Berlin, Heidelberg), pp. 235–249, Springer Berlin Heidelberg, 2005.

[41] L. Shang, Y. Lu, and J. Xue, "Fast and precise points-to analysis with incremental cfl-reachability summarisation: Preliminary experience," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, (New York, NY, USA), pp. 270–273, ACM, 2012.

[42] Y. Lu, L. Shang, X. Xie, and J. Xue, "An incremental points-to analysis with cfl-reachability," in *Proceedings of the 22Nd International Conference on Compiler Construction*, CC'13, (Berlin, Heidelberg), pp. 61–81, Springer-Verlag, 2013.

[43] T. Szabó, S. Erdweg, and M. Voelter, “Inca: A dsl for the definition of incremental program analyses,” in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, (New York, NY, USA), pp. 320–331, ACM, 2016.

[44] Z. Ujhelyi, G. Bergmann, Á. Hegedüs, Á. Horváth, B. Izsó, I. Ráth, Z. Szatmári, and D. Varró, “Emf-incquery: An integrated development environment for live model queries,” *Sci. Comput. Program.*, vol. 98, pp. 80–99, 2015.

[45] B. Hardekopf and C. Lin, “The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code,” in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 290–299, 2007.

[46] J. A. La Poutré and J. van Leeuwen, “Maintenance of transitive closures and transitive reductions of graphs,” in *Proceedings of the International Workshop WG ’87 on Graph-theoretic Concepts in Computer Science*, (New York, NY, USA), pp. 106–120, Springer-Verlag New York, Inc., 1988.

[47] G. Bergmann, I. Ráth, T. Szabó, P. Torrini, and D. Varró, “Incremental pattern matching for the efficient computation of transitive closure,” in *Graph Transformations* (H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, eds.), (Berlin, Heidelberg), pp. 386–400, Springer Berlin Heidelberg, 2012.

[48] R. Tarjan, “Depth first search and linear graph algorithms,” *SIAM JOURNAL ON COMPUTING*, vol. 1, no. 2, 1972.

[49] M. A. Bender, J. T. Fineman, S. Gilbert, and R. E. Tarjan, “A new approach to incremental cycle detection and related problems,” *ACM Trans. Algorithms*, vol. 12, pp. 14:1–14:22, Dec. 2015.

[50] S. based IR in WALA. `https://github.com/wala/WALA/wiki/Intermediate-Representation-(IR)`, 2018.

[51] B. Livshits, "Improving software security with precise static and runtime analysis," 2006. AAI3242585.

[52] Y. Li, T. Tan, and J. Xue, "Understanding and analyzing java reflection," *CoRR*, vol. abs/1706.04567, 2017.

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

[54] O. Lhoták and L. Hendren, "Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation," *ACM Trans. Softw. Eng. Methodol.*, vol. 18, pp. 3:1–3:53, Oct. 2008.

[55] M. Sridharan and R. Bodík, "Refinement-based context-sensitive points-to analysis for java," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, (New York, NY, USA), pp. 387–400, ACM, 2006.

[56] M. Sridharan, D. Gopan, L. Shan, and R. Bodík, "Demand-driven points-to analysis for java," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, (New York, NY, USA), pp. 59–76, ACM, 2005.

[57] T. Reps, "Undecidability of context-sensitive data-dependence analysis," *ACM Trans. Program. Lang. Syst.*, vol. 22, pp. 162–186, Jan. 2000.

[58] B. Hardekopf and C. Lin, "Semi-sparse flow-sensitive pointer analysis," in *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, (New York, NY, USA), pp. 226–238, ACM, 2009.

[59] B. Hardekopf and C. Lin, “Flow-sensitive pointer analysis for millions of lines of code,” in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO ’11, (Washington, DC, USA), pp. 289–298, IEEE Computer Society, 2011.

[60] A. De and D. D’Souza, “Scalable flow-sensitive pointer analysis for java with strong updates,” in *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP’12, (Berlin, Heidelberg), pp. 665–687, Springer-Verlag, 2012.

[61] L. Li, C. Cifuentes, and N. Keynes, “Boosting the performance of flow-sensitive points-to analysis using value flow,” in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE ’11, (New York, NY, USA), pp. 343–353, ACM, 2011.

[62] D. Saha and C. R. Ramakrishnan, “A local algorithm for incremental evaluation of tabledÂălogic programs,” in *Logic Programming* (S. Etalle and M. Truszczyński, eds.), (Berlin, Heidelberg), pp. 56–71, Springer Berlin Heidelberg, 2006.

[63] M. Hind, “Pointer analysis: Haven’t we solved this problem yet?,” in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE ’01, (New York, NY, USA), pp. 54–61, ACM.

[64] M. Emami, R. Ghiya, and L. J. Hendren, “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*, PLDI ’94, (New York, NY, USA), pp. 242–256, ACM, 1994.

[65] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras, “Introspective analysis: Context-sensitivity, across the board,” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’14, (New York, NY, USA), pp. 485–495, ACM, 2014.

[66] G. Kastrinis and Y. Smaragdakis, "Hybrid context-sensitivity for points-to analysis," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, (New York, NY, USA), pp. 423–434, ACM, 2013.

[67] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to analysis for java," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, pp. 1–41, Jan. 2005.

[68] J. Whaley and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, (New York, NY, USA), pp. 131–144, ACM, 2004.

[69] Y. Sui, S. Ye, J. Xue, and P.-C. Yew, "Spas: Scalable path-sensitive pointer analysis on full-sparse ssa," in *Proceedings of the 9th Asian Conference on Programming Languages and Systems*, APLAS'11, (Berlin, Heidelberg), pp. 155–171, Springer-Verlag, 2011.

[70] D. J. Pearce, P. H. Kelly, and C. Hankin, "Efficient field-sensitive pointer analysis of c," *ACM Trans. Program. Lang. Syst.*, vol. 30, Nov. 2007.

[71] N. Heintze and O. Tardieu, "Demand-driven pointer analysis," in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, (New York, NY, USA), pp. 24–34, ACM, 2001.

[72] Y. Smaragdakis and G. Balatsouras, "Pointer analysis," *Found. Trends Program. Lang.*, vol. 2, pp. 1–69, Apr. 2015.

[73] M. Sridharan and S. J. Fink, "The complexity of andersen's analysis in practice," in *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09,

(Berlin, Heidelberg), pp. 205–221, Springer-Verlag, 2009.

[74] J. Dietrich, N. Hollingum, and B. Scholz, "A note on the soundness of difference propagation," in *Proceedings of the 18th Workshop on Formal Techniques for Java-like Programs*, FTfJP'16, (New York, NY, USA), pp. 3:1–3:5, ACM, 2016.

[75] W. Landi and B. G. Ryder, "Pointer-induced aliasing: A problem classification," in *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, (New York, NY, USA), pp. 93–103, ACM, 1991.

[76] A. Gupta, I. S. Mumick, and V. S. Subrahmanian, "Maintaining views incrementally," in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, (New York, NY, USA), pp. 157–166, ACM, 1993.

[77] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, (New York, NY, USA), pp. 243–262, ACM, 2009.

[78] M. Bravenboer and Y. Smaragdakis, "Exception analysis and points-to analysis: Better together," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, (New York, NY, USA), pp. 1–12, ACM, 2009.

[79] N. Grech and Y. Smaragdakis, "P/taint: Unified points-to and taint analysis," *Proc. ACM Program. Lang.*, vol. 1, pp. 102:1–102:28, Oct. 2017.

[80] Y. A. Liu and S. D. Stoller, "From datalog rules to efficient programs with time and space guarantees," *ACM Trans. Program. Lang. Syst.*, vol. 31, pp. 21:1–21:38, Aug. 2009.

[81] K. T. Tekle and Y. A. Liu, "Precise complexity guarantees for pointer analysis via datalog with extensions," *TPLP*, vol. 16, pp. 916–932, 2016.

[82] B. Motik, Y. Nenov, R. Piro, and I. Horrocks, "Incremental update of datalog materialisation: The backward/forward algorithm," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, pp. 1560–1568, AAAI Press, 2015.

[83] S. Putta and R. Nasre, "Parallel replication-based points-to analysis," in *Proceedings of the 21st International Conference on Compiler Construction*, CC'12, (Berlin, Heidelberg), pp. 61–80, Springer-Verlag, 2012.

[84] M. Méndez-Lojo, A. Mathew, and K. Pingali, "Parallel inclusion-based points-to analysis," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, (New York, NY, USA), pp. 428–443, ACM, 2010.

[85] G. System. `http://iss.ices.utexas.edu/`, 2017.

[86] M. Mendez-Lojo, M. Burtscher, and K. Pingali, "A gpu implementation of inclusion-based points-to analysis," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, (New York, NY, USA), pp. 107–116, ACM, 2012.

[87] V. Nagaraj and R. Govindarajan, "Parallel flow-sensitive pointer analysis by graph-rewriting," in *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, (Piscataway, NJ, USA), pp. 19–28, IEEE Press, 2013.

[88] J. Zhao, M. G. Burke, and V. Sarkar, "Parallel sparse flow-sensitive points-to analysis," in *Proceedings of the 27th International Conference on Compiler Construction*,

CC 2018, (New York, NY, USA), pp. 59–70, ACM, 2018.

[89] M. Edvinsson, J. Lundberg, and W. Löwe, "Parallel points-to analysis for multi-core machines," in *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, (New York, NY, USA), pp. 45–54, ACM, 2011.

[90] Y. Su, D. Ye, and J. Xue, "Parallel pointer analysis with cfl-reachability," in *Proceedings of the 2014 Brazilian Conference on Intelligent Systems*, BRACIS '14, (Washington, DC, USA), pp. 451–460, IEEE Computer Society, 2014.

[91] J. Dietrich, N. Hollingum, and B. Scholz, "Giga-scale exhaustive points-to analysis for java in under a minute," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, (New York, NY, USA), pp. 535–551, ACM, 2015.

[92] T. J. Marlowe and B. G. Ryder, "An efficient hybrid algorithm for incremental data flow analysis," in *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, (New York, NY, USA), pp. 184–196, ACM, 1990.

[93] Z. Lai, S. C. Cheung, and W. K. Chan, "Detecting atomic-set serializability violations in multithreaded programs through active randomized testing," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, (New York, NY, USA), pp. 235–244, ACM, 2010.

[94] J. Huang and C. Zhang, "An efficient static trace simplification technique for debugging concurrent programs," in *Proceedings of the 18th International Conference on Static Analysis*, SAS'11, (Berlin, Heidelberg), pp. 163–179, Springer-Verlag, 2011.

[95] J. Huang and C. Zhang, "Lean: Simplifying concurrency bug reproduction via replay-supported execution reduction," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, (New York, NY, USA), pp. 451–466, ACM, 2012.

[96] "Akka Cluster Usage." `http://http://doc.akka.io/docs/akka/current/java/cluster-usage.html`.

[97] "Titan Graph Partitioning." `http://s3.thinkaurelius.com/docs/titan/0.5.0/graph-partitioning.html`.

[98] 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, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, (New York, NY, USA), pp. 169–190, ACM Press, Oct. 2006.