CONTINUOUS REASONING OF LARGE COMPLEX SOFTWARE VIA INCREMENTAL ANALYSIS

A Dissertation

by

BOZHEN LIU

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

DOCTOR OF PHILOSOPHY

August 2023

Major Subject: Computer Science

ABSTRACT

Nowadays, the number of code bases are growing explosively together with their size and complexity, where million lines of code become common to see. To guarantee the correctness and quality of software applications, program analysis techniques have been used to detect bugs and errors each time when there is a new update. However, most existing techniques are designed for whole program analysis, which become expensive and unscalable as the size of programs consistently increase. Moreover, those techniques are used during the late phase of software development, *e.g.*, testing or production, which may be too late to fix a bug or too difficult to understand a bug.

In this dissertation, we introduce several incremental algorithms which are efficient, precise, sound and scalable for real-world, large programs. Firstly, we present IPA, an incremental context-insensitive pointer analysis for Java programs that handles both statement addition, deletion and modification efficiently. Inspired by new properties we observed, IPA avoids redundant computation and expensive graph reachability analysis from existing approaches, as well as preserves precision as the corresponding whole program analysis.

Secondly, we present SHARP, an incremental algorithm that can be generalized to the most commonly used context-sensitive pointer analysis algorithms, *i.e.*, *k-CFA* and *k-obj*. We propose a precompute algorithm to avoid redundant computation from naively applying IPA on context-sensitive call graph and pointer assignment graph. We also discuss different parallel scenarios for incremental code changes according to GitHub commits, summarize their efficiency, redundancy and conflict, and conclude our parallel algorithm. More importantly, both IPA and SHARP can be easily extended to other programming languages.

Thirdly, we present D4, a static program analysis framework to detect concurrency bugs (*i.e.*, data race and deadlock) during development phase. Except for IPA, D4 is powered by another three incremental algorithms of static happens-before analysis, lock-dependency analysis and static concurrency bug detection. D4 can pinpoint concurrency bugs after a code change, which is several orders of magnitude faster than the corresponding whole program detection.

Finally, we present O2, a new system for detecting data races in complex multi-threaded and event-driven applications. O2 is powered by *origin*, which unifies the concepts of thread and event by an entry point and its attributes. We apply origins on pointer analysis to identify objects that are shared by different origins, which concludes our origin-sharing analysis. We also leverage the result of origin-sensitive pointer analysis in static data race detection, which shows a significant improvement in both precision and scalability.

ACKNOWLEDGMENTS

Completing a PhD is difficult, but the seven and half years journey is really precious to me. I cannot achieve this accomplishment without many amazing people. I apologize in advance to all the wonderful people who I forgot to mention here.

First and foremost, I appreciate Jeff Huang decided to be my advisor when I was a layman in Compute Science. It is impossible for me to go through many difficulties in research and life without his generous guidance, help and support.

I would like to greatly thank my committee members, Riccardo Bettati, Dilma Da Silva, Paul Gratz, who contributed a great deal to my development as a computer scientist. I especially appreciate the guidance from Dr. Da Silva and Dr. Lawrence Rauchwerger for my paper submission and interviews. I also would like to thank Chia-Che Tsai who helps with my paper and prelim. I also thank all the help from our department and graduate office: Scott Schaefer, John Keyser, Hank Walker, and Karrie Bourquin.

My gratitude also goes to my research group members, previous ASER and current O2 lab, which includes but not limit to: Shiyou Huang, Gang Zhao, Bowen Cai, Peiming Liu, Brad Swain, Yanze Li, Qiuping Yi, Siwei Cui, Yahui Sun, Luochao Wang, Shanshan Li, Yifei Shen, Yiqing Zhao, and Seyyed Ali Ayati. It was my pleasure to know you all and work with you through these years.

Finally, I want to thank my family for the understanding and support.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Professor Jeff Huang [advisor], Dilma Da Silva, Riccardo Bettati of the Department of Computer Science and Engineering and Professor Paul Gratz of the Department of Electrical and Computer Engineering.

All other work conducted for the dissertation was completed by the student independently.

Funding Sources

Graduate study was supported by either Graduate Research Assistant or Graduate Teaching Assistant from the Department of Computer Science and Engineering, Texas A&M University.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Software is everywhere in our daily life: around 36K mobile apps were released every month in 2022 through the Apple App Store [5] and 3,739 apps are new to the Play Store every day [6]. This is only for smart phones, not to mention applications running on computers, clusters, wearable devices, and even on cloud. The increasing number of software requirements forces the size of programs to grow rapidly. As of January 2015, Google had 2 billion lines of code (LOC) including all its internet services. For the past eight years, it is consistently adding new services and updates to the existing programs, which further bloats the size of Google code base. This scenario applies on all code bases, *e.g.*, Linux kernel includes 350X more lines of code from V1.0 to V6.0 within 29 years [7].

Not only the size but also the complexity of modern software are growing enormously. The architecture of software changes from monolithic applications to multiple microservices containing multiple services, from single-threaded to multi-threaded interacting with events and messages, from local servers to cloud. This has significantly increased the complexity of the whole code base and the overall development process. To offer better functionality and usability, developers have proposed open-source frameworks, libraries and cloud-based services, which however making software more complex than ever.

The more line of code, the higher the complexity, and the more challenging to maintain the correctness and quality of a program. Hence, bugs and errors are easy to introduce but hard to detect. IBM wasted 62m dollars building an AI system to fight against cancer, however, provided multiple unsafe and wrong treatment recommendations [8]. A front-runner attack leverages race conditions among transactions to steal 280 million dollars on Ethereum [9]. The autonomous driving system from Uber even killed a lady due to a software bug [10]. We cannot afford this cost of huge economic loss or even human lives, thus many research techniques have been developed to analyze programs and detect bugs each time when there is a new upgrade. These techniques are mostly designed for late phases of software development, *e.g.*, testing or production. Hence, it is hard to scale them to large software because the whole program has to be analyzed. Moreover, it may be too late to fix a detected bug, or too difficult to understand a reported problem because the developer may have forgotten the coding context to which the bug pertains.

A promising solution is incremental program analysis. Whenever there is some code change, we save time by only recomputing those outputs which depend on the changed code. When an incremental algorithm is successful, it can be significantly faster than recomputing new outputs naively. Among different techniques, pointer analysis is a fundamental static program analysis technique, which is widely used in compiler optimization [11, 12, 13], to parallel code [14, 15, 16, 17], detect bugs and errors [18, 19, 20]. Incremental pointer analysis [21, 22, 23, 24, 25, 26, 27, 28] has been researched for decades, however, it is still challenging to design such an incremental algorithm due to the consistently increasing size and complexity of modern software. Most existing incremental techniques have their own limitations, *e.g.*, redundant computation, expensive graph reachability analysis, precision loss by assuming static call graph, and hard to scale or parallel.

1.1 Dissertation Contributions

This dissertation presents two contributions to overcome the limitations of existing incremental pointer analysis algorithms: two end-to-end incremental pointer analysis with dynamic update of strong connected component (SCC) for context-insensitive and -sensitive algorithms respectively, and parallel version of our incremental algorithms. Besides, we also contribute two pointer analysis applications that achieve precision, efficiency and scalability on static concurrency bug detection: a fast concurrency analysis framework, D4, that detects concurrency bugs statically and interactively in the programming phase by only considering code changes, and a new system powered with origins for whole program data race detection in complex multi-threaded and event-driven programs. We briefly elaborate our algorithms and applications in the following subsections.

1.1.1 Incremental Pointer Analysis

Our incremental pointer analysis, IPA, is an end-to-end algorithm that can handle statement additions, deletions and modifications. We specially design the algorithm for handling deletion, which is inspired by two new properties (*i.e.*, local neighbours properties) we observed on pointer assignment graph (PAG). This allows us to update a points-to set lazily after we confirm its change for a statement deletion. This is also the reason why our algorithm can significantly outperform the state-of-the-art techniques by avoiding redundant computations and expensive graph reachability analysis. Most importantly, we prove that the output of running IPA on an incremental code change is the same as the one of running the corresponding whole program analysis on the new program by including the change.

The properties and IPA are built on acyclic graphs, which requires to always maintain an acyclic PAG according to incremental changes. Hence, we propose an incremental algorithm to dynamically update the SCCs upon incremental code changes.

IPA is designed for context-insensitive pointer analysis, and also works for context-sensitive algorithms but with redundant computations. We present SHARP that eliminates the redundant computation by conducting a precomputation process and identifying all invalid elements in both call graph (CG) and PAG. SHARP can be applied to any *k-limiting* based context-sensitive algorithms, *e.g.*, *k-CFA* and *k-obj*.

1.1.2 Parallel Incremental Pointer Analysis

To further improve the performance, we present parallel algorithms for our incremental algorithms, IPA and SHARP, respectively. The parallelization for IPA is based on change idempotency property we observed on both context-insensitive and -sensitive PAGs. This property enables us to propagate a points-to set change from a PAG node along its different outgoing PAG edges in parallel with no conflict and redundancy.

We further discuss the parallel scenarios for incremental code changes according to a GitHub commit in SHARP. The discussion focuses on efficiency, redundancy and conflict when we handle intraprocedural and interprocedural deletions (or additions) in parallel. We prove that it is sound with no redundancy and conflict to handle one statement deletion per time from all methods in parallel. So do incremental statement additions.

1.1.3 Fast Static Concurrency Bug Detection via Incremental Analysis

Concurrency bug is one of the most annoying bugs, which is difficult to detect, debug and fix. Many research techniques [29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39] has been developed on detecting concurrency bugs, however, they are designed to analyze the whole program and requires hours or even days to complete. Sometimes, we can't afford to recompute the output each time the code changes.

To address the difficulty of detecting concurrency bugs on large, real-world programs, we present D4, a static framework to detect concurrency bugs during the programming phase. Whenever a developer saves his/her code on IDE, D4 will be triggered to detect concurrency bugs (*e.g.*, data races and deadlocks) according to the code change. D4 is powered by IPA and another three incremental algorithms in static happens-before analysis, lock-dependency analysis and static concurrency bug detection. D4 can pinpoint concurrency bugs introduced by a code change, which outperforms the state-of-the-art incremental race detector, ECHO [27].

1.1.4 Efficient and Precise Static Data Race Detection Through Origins

One major disadvantage of static analysis techniques is the presence of false positive, especially for concurrency bugs that are hard to debug and confirm. To improve the precision of static concurrency bug detection, we present *origins*, a context that unifies the concepts of threads and events by treating them as entry points with a set of data pointer attributes. We present origin-sensitive pointer analysis by adopting origins as context, which precisely identifies shared- and local-memory accesses by different threads and events. We present origin-sharing analysis that precisely compute heap objects local to each origin and objects shared by different origins. We present O2, a whole program static data race detection equipped by origins and three sound optimizations to achieve scalability on million lines of code. Moreover, we have detected 40 new races from mature, real-world, large C/C++, Java and Android projects.

1.2 Outline of Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 introduces the background knowledge and related works on pointer analysis for whole program and incremental code changes, as well as static concurrency bug detection that depends on pointer analysis. Chapter 3 presents our efficient incremental pointer analysis, IPA, for context-insensitive on-the-fly algorithm. Chapter 4 presents our incremental pointer analysis, SHARP, for context-sensitive algorithms, especially generalized for *k-CFA* and *k-obj*. Chapter 5 presents our static concurrency bug detection framework, D4, designed to provide instant feedback to developers during the programming phase. Chapter 6 presents our new system powered by origin-sensitive analyses, which significantly improve the performance and precision of static data race detection for large, real-world programs. Chapter 7 concludes this dissertation and discusses future work.

2. BACKGROUND AND RELATED WORKS [*]

This section introduces the background of pointer analysis and static concurrency bug detection. Section 2.1 presents the basic concepts and algorithms of pointer analysis, as well as existing incremental and parallel techniques with the discussion of their limitations. Section 2.2 presents the previous approaches of static concurrency bug detection with their pros and cons.

2.1 Pointer Analysis

Pointer analysis (often used interchangeably with alias analysis and points-to analysis) is a fundamental static program analysis technique, which computes an over-approximation of the set of objects that a pointer, reference or expression can refer to. This set is called *points-to set*. Each computed points-to set contains all objects the variable may point to at runtime. Many pointer analysis algorithms are based on Andersen's algorithm [40], which computes an inclusion-based constraint. Another classical algorithm is the Steensgaard's algorithm [41], which adopts a set equality constraint. It has the linear time complexity, but is much less precise than the Andersen's algorithm. The result of pointer analysis is always represented by a graph which is pointer assignment graph (PAG) [42] or points-to graph. Both algorithms are imprecise because they assume the static call graph (CG) of a program is available. To be specific, CG is constructed ahead of PAG as shown in Figure 2.1(a), which assume every pointer can point to any object compatible with its declared type.

To improve the precision, Andersen's analysis has been extended to build the inclusion-based constraints dynamically (a.k.a. *on-the-fly*) while constructing the call graph (CG) [42, 43]. As shown in Figure 2.1(b), an iterative process starts with the initially reachable methods (*e.g.*, the main function) and generate pointer assignment edges, points-to sets and call edges with new

[*]Reprinted with permission from "Rethinking Incremental and Parallel Pointer Analysis" by Bozhen Liu and Jeff Huang. ACM Transactions on Programming Languages and Systems, Volume 41, Issue 1, Article No.: 6, pp 1–31, Copyright 2018 by Bozhen Liu. Reprinted with permission from "When Threads Meet Events: Efficient and Precise Static Race Detection with Origins" by Bozhen Liu and Peiming Liu and Yanze Li and Chia-Che Tsai and Dilma Da Silva and Jeff Huang. PLDI 2021: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, June 2021, Pages 725–739, Copyright 2021 by Bozhen Liu.

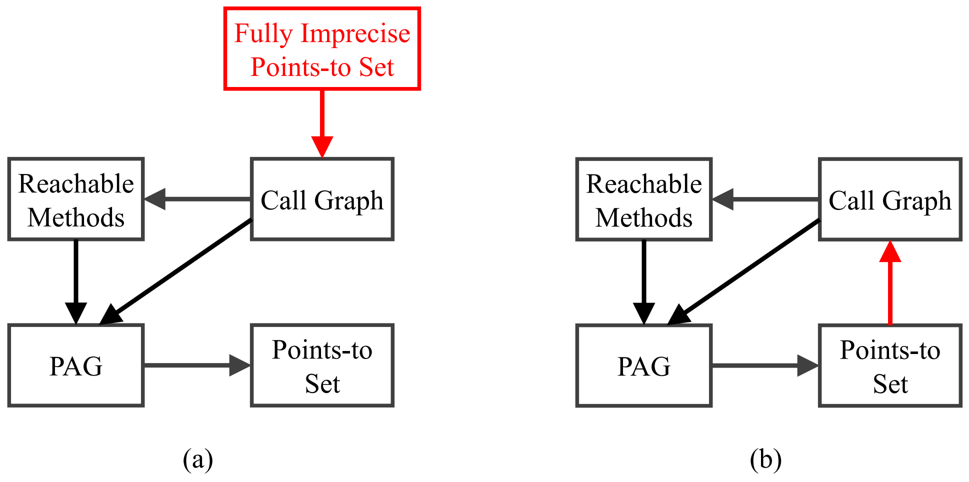


Figure 2.1: The workflows of constructing call graph ahead (a) and on-the-fly (b). Reprinted with permission from [1].

Algorithm 1: On-the-fly Andersen's Algorithm

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

reachable methods, which continues until reaching a fixed point. The detailed algorithm for Java is shown in Algorithm 1, where each iteration consists of two steps:

- Resolve new method call targets based on the current points-to result (*i.e.*, the points-to set of the receiver variable in the method call statements) and extract new points-to constraints for the newly discovered method calls.
- Evaluate the new constraints by following Andersen's analysis in Table 2.1 to compute the points-to set of each variable and discover new call targets if available.

Table 2.1 describes the rules of creating CG and PAG according to the on-the-fly Andersen's al-

Table 2.1: On-the-fly Andersen's Algorithm for Java Programs.

Type		Statement	Pointer Assignment Edge	Constraint
NEW		$x = new\ C()$	$o \to x$	$o_c \in pts(x)$
ASSIGN		$x = y$	$y \to x$	$pts(y) \subseteq pts(x)$
LOAD	FIELD	$x = y.f$	$y.f \rightsquigarrow x$; $\forall o \in pts(y)$: $o.f \to x$	$\forall o \in pts(y)$: $pts(o.f) \subseteq pts(x)$
	ARRAY	$x = y[i]$	$y.* \rightsquigarrow x$; $\forall o \in pts(y)$: $o.* \to x$	$\forall o \in pts(y)$: $pts(o.*) \subseteq pts(x)$
STORE	FIELD	$x.f = y$	$y \rightsquigarrow x.f$; $\forall o \in pts(x)$: $y \to o.f$	$\forall o \in pts(x)$: $pts(y) \subseteq pts(o.f)$
	ARRAY	$x[i] = y$	$y \rightsquigarrow x.*$; $\forall o \in pts(x)$: $y \to o.*$	$\forall o \in pts(x)$: $pts(y) \subseteq pts(o.*)$
INVOKE		$x = y.g(..., a_i, ...)$ //called from $m'()$	$y \overset{invoke\ g()}{\rightsquigarrow}$; $\forall o \in pts(y)$: $m = dispatch(o, g)$ **add call edge** $\{m' \rightarrowtail m\}$, $y \to this_m$, $a_i \to p_i$, $r_m \to x$	$pts(y) \subseteq pts(this_m)$ $pts(a_i) \subseteq pts(p_i)$ $pts(r_m) \subseteq pts(x)$

gorithm for Java programs. There are five types of statements that determine the points-to relation. NEW and ASSIGN are simple statements which only introduce pointer assignment edges (denoted $\to$). An abstract heap object is represented by an allocation node o, and a variable is represented by a variable node x or y. An pointer assignment edge $y \to x$ indicates the inclusion-based constraint between the points-to sets of y and x (*i.e.*, $pts(y) \subseteq pts(x)$). LOAD, STORE and INVOKE are complex statements due to the *dynamic edges* (denoted $\rightsquigarrow$), of which pointer assignment edges vary with the points-to set of its base variable (*i.e.*, y for LOAD and INVOKE, x for STORE). A field reference node $y.f$ represents a pointer dereference, and a concrete field node $o.f$ represents the field f created at the allocation site o.

The rules for array load/store are similar to the ones for field load/store: each array is abstracted with a single field $*$ and considering all array accesses as to that single field. This abstraction is most commonly adopted in pointer analysis algorithms to balance the precision and performance, which do not distinguish different array indexes.

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

The above rules are context-insensitive, flow-insensitive, path-insensitive and field-sensitive, which are simple and commonly used in practice. There are many other dimensions in pointer analysis [44], which has been extensively researched for decades aiming to balance the trade-off between performance and precision, *e.g.*, context-sensitivity [45, 46, 47, 48, 49, 50], flow-sensitivity [51, 52, 53, 54], path-sensitivity [55], field-sensitivity [56, 48], demand-driven [57, 58, 59], algorithmic complexity analysis [60, 61, 62, 63], as well as incremental pointer analysis [22, 23, 24, 25, 26]. In this section, we focus on discussing existing works related to incremental, parallel algorithms, SCC optimizations and thread-/event-related context-sensitivity.

2.1.1 Incremental Algorithms

An incremental code change in a program includes statement additions, deletions and modifications (*i.e.*, a statement is replaced by one or several statements). Hence, an incremental algorithm has to handle all the three scenarios comprehensively. Handling additions is easy according to the on-the-fly Andersen's algorithm described in Algorithm 1. New points-to constraints can be first extracted from the added statement and then provided as the input to run the on-the-fly algorithm.

The difficulty lies in the handling of deletions. For deletion, one has to maintain provenance information on how facts are derived. When a statement is deleted, one has to delete all facts that are "no longer reachable" from existing statements through the provenance information. Consider three consecutive code changes:

1 Add a statement `b=a`

2 Add another statement `c=b`

3 Delete the first statement `b=a`

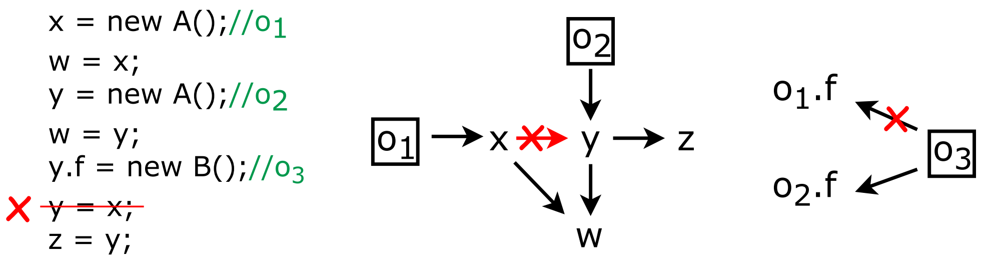


Figure 2.2: An example of edge deletion in a PAG. Reprinted with permission from [1].

When `b=a` is inserted, $pts(b)$ is updated to $pts(b) \cup pts(a)$. When `c=b` is inserted, similarly, $pts(c)$ is updated to $pts(c) \cup pts(b)$. However, when `b=a` is deleted, not only the change in $pts(b)$ should be reversed, but also that the change in $pts(c)$ should be recomputed, because $pts(c)$ was previously updated based on $pts(b)$.

In general, there are two types of existing incremental algorithms for handling deletion: *reset-recompute* [23, 24, 25, 26] and *reachability-based* [22, 27]. However, both of them suffer from performance limitations, and we will illustrate the detail in the following sections.

2.1.1.1 Reset-Recompute Algorithm

Upon a deletion, a simple algorithm is to first reset the points-to sets of all variables that are "*relevant*" to the deleted statement and then recompute them following the same rationale as the on-the-fly algorithm. Here, "relevant" means "reachable" from the root variable of the change in the PAG. Specifically, one can first remove all edges related to the deleted statement from the PAG, and reset (set to empty) the points-to sets of their destination nodes as well as all nodes that these destination nodes can reach (because the points-to sets of all those nodes may be affected). Then, for all the reset nodes, extract their associated points-to constraints and rerun the fixed-point computation following Algorithm 1.

Consider an example in Figure 2.2, in which an edge $x \rightarrow y$ is deleted from the PAG (*e.g.*, due to the deletion of a statement $y = x$ in the program). The root variable of the change is y, since its points-to set may be changed immediately because of the edge deletion. The reset-recompute algorithm first resets $pts(y)$ as well as $pts(z)$ and $pts(w)$ to empty (because z and w are reachable from y). Then it extracts the points-to constraints $pts(y)$= $pts(y) \cup \{o_2\}$, $pts(z)$= $pts(z) \cup pts(y)$, $pts(w)$= $pts(w) \cup pts(y)$, and $pts(w)$= $pts(w) \cup pts(x)$, from the four edges connected to the three reset nodes, *i.e.*, $o_2 \to y$, $y \to z$, $y \to w$ and $x \to w$, and recomputes $pts(y)$, $pts(z)$ and $pts(w)$ until reaching a fixed point. The final values of the points-to sets are: $pts(x)$={o_1}, $pts(y)$={o_2}, $pts(z)$={o_2} and $pts(w)$={o_1, o_2}. As o_1 has been removed from $pts(y)$ and y is the base variable of $y.f$, it also deletes the edge $o_3 \to o_1.f$ and recomputes $pts(o_1.f) = \emptyset$.

The reset-recompute algorithm can be traced back to the DRed algorithm [21] and the incremental pointer aliasing analysis [64]. It is inefficient because most computations on the points-to sets of the reset nodes may be redundant. For example, both before and after the statement deletion, $pts(w)$ remains the same. We call such recomputation on $pts(w)$ as *redundant computation* in the incremental analysis.

Context-Insensitive Many research works have been conducted to reduce the redundant computation for reset-recompute algorithm. Saha et al. [23, 28] propose an incremental and demand-driven points-to analysis based on the DRed algorithm [21]: it marks the affected answers, checks the marked answers, and removes the answers that cannot be rederived. To reduce the redundant checks, a *support graph* is introduced where nodes are the answers, supports and facts, and edges indicate the points-to relations among them. A primary support, which is independent from its answer, is maintained to optimize the marking process. When there is an incremental change, they only mark an answer if its primary support is marked, and then mark all the supports that uses the answer. After the marking stage, if a marked answer has other unmarked supports, they consider the answer is rederived and recursively remove the marks generated from the answer. Otherwise, a recomputation has to be performed on the answer. Instead of PAG, this technique adopts a *support graph* to represent the points-to relations among variables. However, such a graph requires maintaining primary supports for each answer. Besides, the "mark-check-remove mark" method redundantly propagates the marks, since it cannot recognize whether an answer should be removed at first glance. Besides, their rederivation technique requires computing *derivation length* to determine the order in which marked answer should be recomputed first. Such an order prohibits the leverage of parallelization. Besides, the handling of dynamic method calls is very imprecise in this technique: all the methods that have the same number and types of parameters as the call site are considered as targets.

Incremental computation has been extensively discussed in the domain of datalog evaluation. There are several pointer analyses formulated using datalog frameworks [65, 66, 67, 68, 69]. However, despite intensive research [21, 70] for optimizing the incremental evaluation of datalog, datalog engines are still inefficient to handle incremental deletion of the pointer analysis facts. In our experience with LogicBlox [71], a state-of-the-art datalog engine that supports incremental updates of Datalog facts, it cannot even finish in six hours for handling a statement deletion in our benchmarks.

Researchers also have applied the reset-recompute algorithm to different dimensions of pointer analysis. Here, we introduce two state-of-the-art approaches, REVISER and IncA.

REVISER [24] proposes an interprocedural, flow- and context-sensitive data-flow analysis based on the IFDS/IDE (Interprocedural Distributive Environment Transformers) framework. Similar to the *reset-recompute* algorithm, REVISER adopts a *clear-and-propagate* strategy to clear and recompute the relative analysis result when necessary. The difference exists in its frameworks, Soot [72]. The IFDS framework formats the data-flow problem to a graph reachability problem. Furthermore, the IDE framework can solve distributive functions along the graph edges.

IncA [26] proposes a domain-specific language (DSL) to incrementally update program analysis result based on graph pattern matching. It uses *pattern functions* to define the relations between program entities in a program analysis (*e.g.*, the execution order of statements in control-flow analysis, the points-to relations of variables in points-to analysis). IncA can be used to perform incremental points-to analysis by introducing a relation $PointsTo(x, y)$ to represent that variable x can point to variable y. For incremental statement changes, IncA initially updates the interprocedural control-flow graph (ICFG) incrementally. Then, according to the new control flow changes, it performs an incremental graph pattern matching (adopted from EMF-IncQuery [73]) to propagate changes to all dependent points-to relations and re-analyze the changed program entities. EMF-IncQuery is an incremental graph query engine to capture and execute live queries over EMF models (such as UML). It also supports incremental updates of model elements based on the Rete networks [74]. However, the specific algorithm for handling deletion is not very clear and may be determined by the considered patterns.

Context-Sensitive To achieve efficiency in context-sensitivity, cloning-based technique [75] creates multiple instances for a method and each links with a distinct context. This can be adapted to solve incrementally added statements, however, it cannot handle code deletion which requires complicate algorithms to guarantee the soundness. Similarly, *diff-graph* [76] represents the accesses of a method in a flow-sensitive, but context-insensitive way in order to reduce the workload of creating a points-to graph [77], which however only handles addition.

Lu et al. [78] present an incremental pointer analysis based on CFL-reachability, where a points-to query is answered by finding the CFL-reachable paths in the PAG from the queried variable to objects.

Many recent techniques [47, 79, 80, 81, 82, 83, 84, 85, 86] focus on selective context-sensitivity, which is another promising direction to scale pointer analysis. The idea is to analyze selective program elements context-sensitively to avoids the combinatorial explosion in analysis time and memory space. The selection relies on heuristics from user inputs or machine learning techniques, or pattern matching of the imprecision in dataflow, which shows significant performance and/or precision improvement.

2.1.1.2 Reachability-Based Algorithm

Before removing an object node from a points-to set, we can always firstly check the path reachability from the object node to the pointer node. In this way, the points-to sets of those nodes that are potentially affected by the deletion are not reset, but are updated lazily only if they are not reachable from the object nodes. This algorithm does not incur any redundant computation on the points-to set. However, it requires repeated whole-graph reachability checking, which is expensive for large PAGs.

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

The main scalability bottleneck of the reachability-based algorithm is that the worst case time complexity for checking path reachability is linear in the PAG size, which can be very large for real-world programs. The performance can be improved by parallelizing the reachability check for different object nodes, however, the time complexity is still linear in the PAG size.

Another important issue of incremental pointer analysis is how to handle the deletion of method call statement. Most existing pointer analysis algorithms for handling dynamic code changes assume a static program call graph [22, 23, 24, 25], which loses precision after adding or deleting a method call statement. Moreover, due to the potential change in a base variable, a statement deletion may cause the change in both PAG and CG, thus it should be carefully handled to guarantee the soundness and correctness.

2.1.2 Parallel Algorithms

Most existing parallel algorithms are designed to speed up the propagation of initial points-to constraints for whole-program pointer analysis, which require a static whole program.

Putta and Nasre [87] 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. [88] 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 [89], the rules are performed in parallel on non-interfering nodes in the constraint graph. This graph rewriting algorithm has been further implemented on GPU [90] with an efficient graph representation for the constraint graph under the GPU memory model. Nagaraj et al. [91] propose a flow-sensitive pointer analysis which is parallelized based on the graph rewriting rules from [88].

PSEGPT [92] is also designed for parallel flow-sensitive pointer analysis, which relies on a new representation that combines points-to relations and def-use chain on heap. It decomposes the points-to analysis into fine-grained units of work that can be implemented in an asynchronous task-parallel programming model. The operations that propagate the points-to information can be executed in parallel if they obey the data dependence among operations.

Edvinsson et al. [93] 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.

Su et al. [94] 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.

2.1.3 SCC Optimizations

An important optimization for pointer analysis in practice is to compute the *strongly-connected component*s (SCCs) in the PAG. Because all nodes in the same SCC are guaranteed to have identical points-to sets, all these nodes can be collapsed into a single node. Hence, the points-to sets of all the nodes in an SCC can be updated all together to leverage the benefit of cycle elimination.

A number of pointer analysis algorithms [50, 95, 96, 97, 98] have adopted some SCC optimizations to improve performance, *e.g.*, Tarjan's algorithm [99]. However, all these optimizations do not update SCCs according to dynamic graph changes. To handle dynamic program changes, SCCs must also be updated dynamically.

La Poutré et al. [100] propose the first algorithm to dynamically maintain the transitive reduction of a directed graph regarding to graph edge additions and deletions, *i.e.*, dynamically collapse/break SCCs. Bergmann et al. [101] adapt the algorithm [100] for incremental graph pattern matching to improve the scalability. Marlowe et al. [102] propose an incremental data flow analysis that can decompose/compose the affected SCCs in the data flow graph whenever there are data flow changes, such that a precise and correct result can be obtained efficiently.

2.1.4 Thread and Event Related Context-Sensitivity

Recently, a new dimension, *thread-sensitivity*, has been proposed in pointer analysis, in order to achieve better precision when applying the result of pointer analysis on static concurrency bug detection. In this section, we will discuss the existing pointer analysis works designed for multi-threaded and event-driven programs.

2.1.4.1 Thread or Event

Researchers have been comparing thread and event models for so long time: Lauer and Needham [103] compared event-driven systems with thread-based systems and regarded threads and events as intrinsically dual to each other; Ousterhout [104] then argued against using threads due to the difficulty of developing correct threaded code; and Lee [105] also noted the lack of understandability and predictability of multi-threaded code due to nondeterminism and preemptive scheduling. On the other hand, Von Behren et al. [106] remarked on the "stack ripping" problem of events and advocated for using threads for their simple and powerful abstraction. In Capriccio [107], they used static analysis and compiler techniques to transform a threaded program into a cooperatively-scheduled event-driven program with the same behavior. Adya et al. [108] also backed Von Behren by noting the question of threads or events as orthogonal to the question of cooperative or preemptive scheduling. Meanwhile, a unified concurrency model at the programming language level becomes more practical and useful, including Scala with Actors [109] and Haskell [110].

2.1.4.2 Unifying Thread and Event

It is highly desirable to unify threads and events so that these two models can be combined to achieve optimal performance on a real environment. In fact, for application domains with both heavy concurrency and intensive I/O such as web and database servers, a hybrid model of threads and events is often used. For example, in web servers and mobile applications, I/O and lifecycle events are used to model network connections and user interactions, and thread pools are used to handle concurrent user requests. Researchers have exploited this direction at the programming language level, including Scala with Actors [109] and Haskell [110], to produce a unified concurrency model.

2.1.4.3 Thread- and Event-Sensitive Algorithms

Many dataflow analysis techniques [111, 112, 113, 114, 115, 116] have been proposed for event-driven programs to model event lifecycles and event handlers, but they only scale to hundreds of lines of code. These techniques either compromise precision due to unsound treatment of thread interactions or lose scalability due to expensive value-flow analysis.

Using threads as the context in pointer analysis is not new [117, 118, 119, 120]. Other algorithms for multithreaded programs are not general as they target specific analyses (*e.g.*, escape analysis and region-based allocation [121], synchronization elimination [122, 123]), or only work for structured multithreading (*e.g.*, Cilk [124, 125]). In addition to using threads or events, prior research has proposed a variety of ways to represent contexts such as *call-site* [126, 127], *receiver object* [49] and *type* [128]. Recently, selective context-sensitive techniques [19, 79, 80, 81, 82, 129, 130, 131] have also been proposed. Although much progress has been made, context-sensitive pointer analysis remains difficult to scale.

2.2 Static Concurrency Bug Detection

As an important application of pointer analysis, much research has been done for detecting bugs statically by leveraging the result of pointer analysis. This section introduces existing works in static concurrency bug detection.

Cheetah [132] is a Just-In-Time static analysis to detect programming errors quickly. Instead of incremental analysis, Cheetah uses a layered analysis which expands the analysis scope gradually from recent changes to code further away. In addition, Cheetah focuses on data-flow analyses such as taint analysis for Android apps instead of concurrency analysis.

RacerD [133] is a practical concurrency error detector developed by Facebook. It relies on code annotation and performs race detection with aggressive ownership analysis, rather than using pointer analysis and happens-before analysis to analyze the impact of a code change.

There exist a wide range of techniques for detecting concurrency bugs in the multithreaded whole program. For example, RacerX [30] and Chord [34] detected many real-world data races and deadlocks in C/C++ and Java programs with static analysis. HARD [134] utilizes hardware features to improve the race detection performance, Fonseca et al. [135] leverage linearizability testing to find concurrency bugs, and ConSeq [136] analyzes sequential memory errors to find concurrency bugs. However, all these techniques focus on whole program analyses which may be difficult to achieve efficiency.

Meanwhile, as the size and complexity of programs increase, there exist various difficulties when applied classic detection techniques to modern software. RacerX contains many heuristics and engineering decisions, which are difficult to duplicate. RELAY [38] depends on the CIL compiler front-end, which supports only a subset of C and has not been actively developed [137]. Technically, RELAY uses a context- and field-insensitive pointer analysis, a major source of false positives. String-pattern-based heuristics are used in RELAY to filter out false aliasing. These heuristics are effective in reducing false positives, but are only specific to the code conventions in the target program and are unsound.

RacerD, developed at Facebook, is by far the most successful static race detector [133]. It is regularly applied to Android apps in Facebook and has flagged over 2500 issues that have been fixed by developers before reaching production [138]. RacerD's design favors reducing false positives over false negatives through a clever syntactical reasoning, but it does not reason about pointers and thus can miss races due to pointer aliases.

Except for multithreaded programs, races also exit in event-driven programs, which has attracted much attention in recent years [139, 140, 141, 142, 143, 144, 145, 146]. Event-based races can be more challenging to detect than thread-based races because most events are asynchronous and the event handlers may be triggered in many different ways. Moreover, the difficulty in detecting event-based races is exacerbated by interactions between threads and events, which are common in real-world software such as distributed systems. The state-of-the-art race detectors [37, 133, 138] do not perform well in detecting event-based races, also due to the large space of casual orders among event handlers and threads.

3. IPA: INCREMENTAL POINTER ANALYSIS [*]

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

3.1 Introduction

Pointer analysis computes the set of objects that a pointer variable can point to at runtime. It is a fundamental technique underpinning virtually all interesting static program analyses (*e.g.*, compiler optimizations and bug detection) and has been the focus of intensive research [40, 46, 47, 48, 49, 50, 57, 60, 61, 63, 62, 96, 147, 148, 149].

Unfortunately, scaling pointer analysis up to large code bases has been extremely challenging. For example, for the classical Andersen's pointer analysis, it typically takes tens of minutes or hours to analyze real-world applications with hundreds of thousand of lines of code [60, 98, 150]. As a result, typically only imprecise pointer analyses are used in production compilers, missing

[*]Reprinted with permission from "Rethinking Incremental and Parallel Pointer Analysis" by Bozhen Liu and Jeff Huang. ACM Transactions on Programming Languages and Systems, Volume 41, Issue 1, Article No.: 6, pp 1–31, Copyright 2018 by Bozhen Liu.

many potential optimization opportunities [151].

Researchers have investigated incremental pointer analysis since nearly two decades ago [64, 152]. These incremental algorithms [22, 23, 24, 25, 26, 28, 27, 78] promise significant performance improvements over the exhaustive pointer analysis because analyzing code changes is often much faster than analyzing the entire code base. For applications in which pointer analysis has to run repeatedly with respect to frequent but *small* program changes, *e.g.*, bug finding in the IDE (Integrated Development Environment) [27] and incremental compilation [153], this is particularly useful because only a small part of the program needs to be analyzed instead of the whole program.

However, existing incremental algorithms (albeit fast in simple cases) are still too slow to be applied in large real-world applications. For instance, in our experiments we find that existing algorithms [27] can take over half an hour to analyze a statement deletion in large applications. Moreover, most existing algorithms [22, 23, 24, 25] assume a pre-built call graph of the program, which does not hold for scenarios where the call graph itself can be modified by the code changes. Furthermore, some algorithms (*e.g.*, [22]) do not preserve precision but compute a less precise result than the exhaustive analysis.

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

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

We implemented an end-to-end incremental pointer analysis, IPA, in the WALA framework [154] based on our new algorithm, and evaluated it extensively on a wide range of real-world large complex Java applications from the DaCapo-9.12 benchmarks [155]. The experimental results show dramatic efficiency and scalability improvements compared to existing algorithms: on a 48-core HPC machine, IPA takes only 24ms on average and 5.5s in the worst case to analyze a change, achieving more than 200X speedups over existing incremental algorithms based on graph reachability and recomputation, and it is two to five orders of magnitude faster than the exhaustive pointer analysis while preserving the precision. We have also applied IPA to detect data races incrementally in the IDE. Through the use of IPA for incremental pointer analysis, the performance of a state-of-the-art incremental race detector [27] is improved by as much as 100X.

To the best of our knowledge, IPA is the first parallel incremental pointer analysis that realizes both the change incrementalism and the algorithmic parallelization. Although previous research has proposed separately a number of incremental algorithms [23, 24, 26] and parallel algorithms [87, 88, 93, 94, 91, 92], none of them is both incremental and parallel, which is challenging to design and implement efficiently.

In summary, we claim the following contributions:

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

- We present an extensive evaluation of IPA on large complex real-world Java programs as well as an application for IDE-based incremental race detection, demonstrating significant performance improvements over existing algorithms.
- IPA is open source [156] and has been integrated into the WALA code base.

3.2 New Incremental Algorithms

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

We first present the two key properties. We then present the basic incremental algorithm in Section 3.2.1 and the parallel incremental algorithm in Section 3.2.2.

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 multiple edge deletions in Section 3.2.1. Based on the transitivity property, we can prove the following lemma:

Lemma 1: Incoming neighbours property. Consider an acyclic PAG and a pointer node q of which an object $o \in pts(q)$. If q has an incoming neighbour r (*i.e.*, there exists an edge $r \rightarrow q$) 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 3.1. First, because $o \in pts(r)$, due to transitivity, o can reach r. Second, because the PAG is acyclic, there cannot exist a path $o \rightarrow \ldots \rightarrow q \rightarrow \ldots \rightarrow r \rightarrow q$ (which contains a cycle).

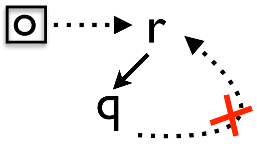


Figure 3.1: Illustration of the incoming neighbours property. Reprinted with permission from [1].

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,

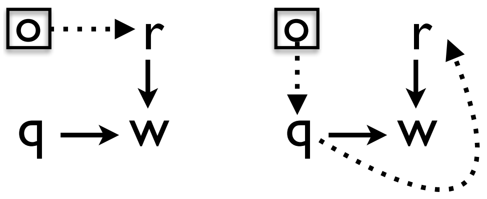


Figure 3.2: Illustration of the outgoing neighbours property. Reprinted with permission from [1].

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

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

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

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

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

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

Consider again the example in Figure 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.

3.2.1 Basic Incremental Algorithm

In Theorems 1 and 2, we have made the assumption that the PAG is acyclic and we have considered only one edge deletion. The acyclic PAG can be satisfied by the SCC optimization, which is known in existing literature for whole program pointer analysis [96]. However, in the incremental setting, the SCCs must be dynamically updated. We first give a brief overview of our incremental SCC detection algorithm, which shares the same main idea with [100, 101]. Based on it, we then present our incremental algorithms for handling edge deletion and addition.

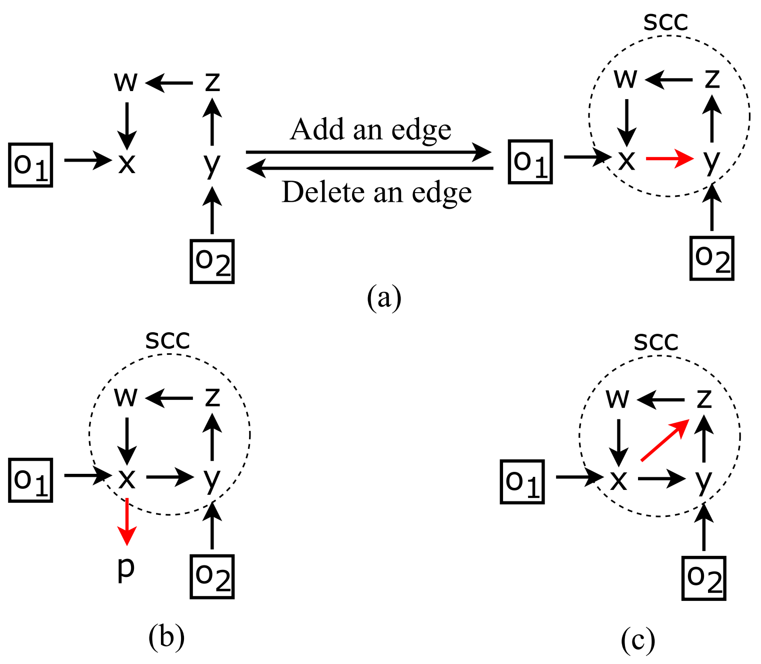


Figure 3.3: Three SCC scenarios. Reprinted with permission from [1].

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.

3.2.1.1 Incremental SCC Detection

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

1. The edge does not belong to any SCC: nothing to do with existing SCCs.

2. The edge belongs to a certain SCC, but deleting the edge does not break the SCC. In this case,

we keep the super node corresponding to the collapsed SCC in the PAG, and only remove the edge from the collapsed SCC. We use Tarjan's linear-time algorithm [99] 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 [157] 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 3.3 illustrates the incremental SCC detection with examples. Figure 3.3(a) shows that adding the edge $x \to y$ creates a new SCC and deleting the edge breaks the SCC. Figure 3.3(b) shows that the edge $x \to p$ does not belong to any SCC, so adding/deleting the edge does not create new SCCs or affect existing SCCs. Figure 3.3(c) shows that the edge $x \to z$ belongs to an SCC, but adding/deleting it does not augment or break the SCC.

3.2.1.2 Incremental Edge Deletion

Algorithm 2 shows our incremental algorithm for handling edge deletion. We maintain a PAG and a worklist, which is initialized to the input deleted edge. In each iteration, one edge from the worklist is processed, which involves two steps. First, we remove the edge from the PAG and handle the SCCs according to the incremental SCC detection algorithm described in Section 3.2.1.1.

Algorithm 2: DeleteEdge(e)

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

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

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

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 $x \to y$, Δ is initialized to $pts(x)$, because after deleting the edge all objects in $pts(x)$ may be removed from $pts(y)$. Then, we check the incoming neighbours of y; if any change in Δ is contained in the points-to set of a neighbour, the change should be skipped, *i.e.*, not applied to $pts(y)$. Hence, we remove from Δ all the objects that overlap with the points-to sets of y's incoming neighbours. For the remaining objects in Δ, we then remove them from $pts(y)$ and propagate them further to all of y's outgoing neighbours.

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

3.2.1.3 Incremental Edge Addition

Algorithm 3 shows our incremental algorithm for handling edge insertion, which follows the on-the-fly algorithm in Algorithm 1. Compared with our incremental deletion algorithm, it has three main differences. First, instead of deleting edges from the PAG, it always adds edges. Sec-

Algorithm 3: AddEdge(e)

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

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

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

ond, 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 *CheckNewAddedEdges* procedure takes a points-to set change Δ and a target node y as input, and returns a list of added PAG edges to the worklist. For each object $o \in \Delta$, for a *load* statement $x = y.f$ we add a new edge $o.f \to x$; for a *store* statement $y.f = x$, we add a new edge $x \to o.f$; and for a *call* statement $y.m()$, we check the method $T.m()$ (where T is the type of the object o) and add the corresponding method call edges, and also analyze the method body if $T.m()$ is new.

3.2.2 Parallel Incremental Algorithm

Our parallel incremental algorithms are based on a strong *change idempotency* property of our basic incremental algorithms described in Section 3.2.1.

Lemma 3: Change idempotency 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: $p \to \ldots \to r_1 \to q$ ($path_1$) and $p \to \ldots \to r_2 \to q$ ($path_2$), respectively, where p is the root change node (the 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.

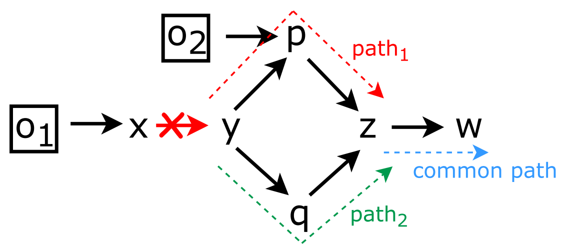


Figure 3.4: An example of parallel change propagation. Reprinted with permission from [1].

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.

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

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

Algorithm 5: ParallelPropagateAddChange(Δ, y)

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

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.

Algorithms 4 and 5 show our parallel incremental algorithms for deletion and insertion, respectively. We propagate the points-to set change of a node along all its outgoing edges in parallel (see line 8 in Algorithm 4 and line 4 in Algorithm 5). Our algorithm guarantees that a change can only propagate through a node at most once, even though there might be multiple parallel propagation paths reaching the same node. Figure 3.4 shows an example. Initially, $pts(y) = pts(q) = \{o_1\}$ and $pts(p) = pts(z) = pts(w) = \{o_1, o_2\}$. After deleting the edge $x \rightarrow y$, $pts(y)$ 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 4, the change propagates along the two paths (*i.e.*, $path_1$ and $path_2$) in parallel, and reaches a common node z. There are three possibilities to consider in this process:

- The propagation along $y \rightarrow p$ completes faster than that along $y \rightarrow q$. At this time, o_1 has been removed from $pts(p)$, and we are still checking the incoming neighbours of q, where $pts(q) = \{o_1\}$). Then, the propagation from $path_1$ reaches z. Since q is an incoming neighbour of z and $o_1 \in pts(q)$, $pts(z)$ will not be changed and hence the propagation from

$path_1$ terminates at z. Later, when the propagation from $path_2$ reaches z, because $pts(q)$ and $pts(q)$ do not contain o_1 anymore, o_1 is finally removed from $pts(z)$ and the change propagates further to w from $path_2$.

- The propagation along $y \rightarrow q$ completes faster than that along $y \rightarrow p$. Opposite to the first case, the change propagation from $path_2$ terminates at z, and the change propagation from $path_1$ continues through z.

- The propagation along both paths reaches p and q at the same time. At this time, $pts(p) = \{o_2\}$ and $pts(q) = \emptyset$. Both changes from p and q propagate to z. No matter which propagation reaches z first, o_1 will be removed from $pts(z)$. When the later propagation comes, no change to $pts(z)$ will be performed, since o_1 has already been removed from $pts(z)$. If both of them reach z at the same time and both attempt to remove o_1 concurrently, to minimize the amount of the points-to set computation, we can synchronize the updates of $pts(z)$, such that only one of them can succeed and can continue the change propagation.

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

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

3.2.2.1 Synchronization-Free Implementation

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

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

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

1. In Algorithm 4, T1:`contains(o)` at line 2 on $pts(r)$ and T2:`remove(o)` at line 7 on $pts(z)$. Consider the operation `contains(o)` by T1. With or without synchronization, it always returns either true or false. If false, then `o` will be removed from $pts(y)$ at line 7 by T1. If true, `o` will not be removed from $pts(y)$ by T1; however, `o` is removed from $pts(z)$ by T2 and because y is an outgoing neighbour of z the change will propagate to y. Finally, `o` will be removed from $pts(y)$ by T2 or by another thread.

2. In Algorithm 4, both T1:`remove(o)` and T2:`remove(o)` at line 7 on $pts(y)$. The entry for o in $pts(y)$ will be set to 0 (meaning o is not included) by both T1 and T2, *i.e.*, `o` will be removed from $pts(y)$.

3. In Algorithm 5, T1:`contains(o)` at line 1 on $pts(y)$ and T2:`add(o)` at line 3 on $pts(y)$. The operation `contains(o)` by T1 may return either true or false. If false, then `o` will be added to $pts(y)$ at line 7 by T1. If true, `o` has already been added to $pts(y)$ by T2.

4. In Algorithm 5, both T1:`add(o)` and T2:`add(o)` at line 3 on $pts(y)$. The entry for o in $pts(y)$ will be set to 1 (meaning o is included) by both T1 and T2, *i.e.*, `o` will be added to $pts(y)$.

3.3 End-to-End Incremental Pointer Analysis

In this section, we present IPA, an end-to-end incremental pointer analysis for real-world Java programs based on our new incremental algorithms described in Section 3.2. The presented pointer analysis is context-, path- and flow-insensitive.

Algorithm 6: Incremental Pointer Analysis for Java

```
Input : $\Delta_{P_{IR}}$ - a set of IR changes.
        Deletions: D: -{d1,d2,...};
        insertions: I: +{i1,i2,...}.
  // for each deleted IR statement
1 foreach $s \in D$ do
      // extract edge(s) according to Table 3.1
2     $e \leftarrow$ ExtractEdge($s$)
      // call Algorithm 2 for each deleted edge
3     DeleteEdge($e$)
4 end
  // for each inserted IR statement
5 foreach $s \in I$ do
      // extract edge(s) according to Table 3.1
6     $e \leftarrow$ ExtractEdge($s$)
      // call Algorithm 3 for each added edge
7     AddEdge($e$)
8 end
```

Table 3.1: Java statements and their corresponding pointer assignment edges. Reprinted with permission from [1].

Statement	Pointer Assignment 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 has performed an initial commit of her project, we compile the project, translate the Java bytecode to an SSA-based IR [158] and use the IR to construct a PAG. Then, the developer commits a collection of program changes $\Delta_{P_{src}}$. We recompile the project with $\Delta_{P_{src}}$ to obtain the updated IR, compare with the old IR to obtain the IR changes $\Delta_{P_{IR}}$, which contains multiple new IR statement insertions and/or old IR statement deletions or modifications[†]. $\Delta_{P_{IR}}$ can be divided into two disjoint sets: D - a set of old IR statement deletions and I - a set of new IR statement insertions. In our implementation, we use the Java Source Front End provided in WALA to translate the changed source file to Eclipse AST and then to IR (which is very fast, typically within 10ms for a changed method). If the source file is uncompilable, *e.g.*, because of syntax error or missing dependencies, DPA will not run.

Algorithm 6 shows our end-to-end algorithm. For each IR statement, we first extract the corresponding edges in the PAG according to Table 3.1. 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.

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 $o.*$.

[†]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.

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.

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

Dynamic Language Features As IPA analyzes code statically, it is difficult to handle dynamic language features such as dynamic class loading and reflection in Java. This is a known challenging problem in static analysis, and previous research has proposed a number of effective techniques to help alleviate this problem, notably [159, 160, 161, 162]. In our implementation, we rely on WALA's existing support to deal with dynamic features. For example, WALA has a significant support for reflective constructs by using the concrete type of the receiver class in a context-sensitive way [163].

3.4 Discussions

3.4.1 Adapting to Context-Sensitive, Flow-Sensitive and Other Problems

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

Context-Sensitivity We note that our incremental algorithms can also apply to context-sensitive pointer analysis because the handling of edge insertions and deletions is orthogonal to the representation of context. In general, there are two types of techniques in context-sensitive pointer analysis: k-CFA [43, 126, 164] and CFL-reachability [22, 48, 59, 78].

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

CFL-reachability pointer analysis also represents a program by a PAG, 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. Although the CFL-reachability paths are not transitive closures, the transitive closure property may still be maintained by using a trace-based incremental CFL-reachability algorithm [78]. The idea is that the computed feasible paths for answering a points-to query can be cached as traces. In the cached traces, the transitive closure property still holds. When a change occurs, we can check the path feasibility of the traces that contain the changed node, and as long as the path is still feasible, we can apply Algorithms 2 and 3 along the cached traces to incrementally update the points-to results.

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

Flow-Sensitivity Flow-sensitive pointer analysis is often a lot more expensive and complicated than its flow-insensitive version. To determine a points-to set at a program point, a partial SSA representation and control flow graph can be combined to infer points-to and def-use relations and to avoid unnecessary propagation, which can also be represented by graphs [51, 52, 53]. Sometimes, the analysis can be cast to a graph reachability problem [54].

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

A possible solution is to decouple the graph that encodes points-to and control flow as proposed in IncA [26], in which points-to graph and control-flow graph are maintained separately. In this case, a two-step update can be performed: first, update the def-use information according to the control flow changes; second, update the points-to relation based on the new def-use and program changes. Hence, our incremental algorithm can be modified to perform on a single graph in each step. However, we note that IncA may loss precision because the construction of a precise control-flow graph and a precise points-to analysis are typically recursively dependent on each other.

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

- G is a directed graph, and SCC collapsing can be applied on G to obtain an acyclic graph, with or without precision loss on the property of interest;
- Each node in V represents a set of elements, and each edge in E represents a constraint which propagates the elements.
- The constraint satisfies the local neighbour properties presented in Section 3.2;
- The direction on an edge indicates the propagation direction;
- The propagation of elements on G can terminate.

For example, dynamic algorithms for directed graph reachability are used in motion planning to search for the next move in the configuration space [165] and to identify collision-free paths [166]. Our algorithms may be applied there to improve efficiency.

3.4.2 Scheduling of Changed Statements

Since multiple statement additions and deletions may happen 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 [97]) can be performed to reduce such redundant work.

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

3.5 Evaluation

We implemented IPA in WALA [154], a popular static program analysis framework that supports a variety of pointer analyses. We modified the *ZeroOneContainerCFA* pointer analysis, which uses a single InstanceKey for every allocation site. To compare with the existing classic algorithms, we also implemented the reset-recompute and the reachability-based algorithms in the same framework. These two algorithms are based on our prior work, ECHO [27], an IDE-based incremental bug detection tool for detecting data races. To compare with the state-of-the-art tools, we also performed an evaluation of REVISER[‡] [24]. We evaluated IPA and REVISER on a collection of 14 real-world large Java applications from DaCapo-9.12, as shown in Table 3.2. However, we

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

note that this is not an apples-to-apples comparison because IPA implements an incremental pointer analysis, while REVISER implements different incremental client analyses over a non-incremental pointer and call-graph analysis. Moreover, REVISER is context- and flow-sensitive, while IPA is not. In this section, we report the results of our experiments.

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

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

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

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

Table 3.2: Benchmarks and the PAG metrics. Reprinted with permission from [1].

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

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

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

Benchmarks The metrics of the benchmarks and their PAGs are reported in Table 3.2. Columns 2-6 report the numbers of classes, methods, pointer nodes, object nodes and edges in the PAGs, respectively. More than half of the benchmarks contain over 1M pointer nodes and over 200M edges in the PAG. The default ZeroOneContainerCFA pointer analysis creates an object node for every allocation site and has unlimited object-sensitivity for collection objects. For all benchmarks certain JDK libraries such as *java.awt.** and *java.nio.** are excluded[§] to ensure that the exhaustive pointer analysis analysis can finish within 6 hours.

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

Table 3.3: Performance of the incremental algorithms for handling deletion. Reprinted with permission from [1].

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

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

3.5.1 Performance of Incremental Deletion

Table 3.3 compares the performance between the exhaustive pointer analysis (*Full*) and the incremental algorithms for deletion. Overall, IPA achieves dramatic speedups over the other algorithms. On average, IPA is two to five orders of magnitude faster than the exhaustive algorithm and two orders of magnitude faster than the other two existing incremental algorithms (*i.e.*, reset-recompute and reachability-based) and REVISER. For most benchmarks, the exhaustive analysis takes several hours to compute (2.4h on average). For a deletion change, the *reset-recompute* algorithm takes 26s on average, the *reachability*-based algorithm takes 39s, whereas *IPA-1* takes only 73ms to analyze, which is at least 300X faster than the other algorithms. REVISER shows an impressive performance (*e.g.*, 153s for all cases on average and 268s for the worst cases), especially for the worst-case scenarios. However, it still two orders of magnitude slower than *IPA-1*. REVISER did not perform a successful run on *jython* (indicated by “-" in Table 3.3 and 3.4), due to an unrecognized Java type in the Jimple IR construction. Compared to *IPA-1*, *IPA-48* further improves performance by an order of magnitude. *IPA-48* takes only 24ms on average, more than three orders of magnitude faster than existing incremental algorithms.

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

The worst case scenarios are often hundreds to thousands of times slower than the average cases in the two incremental algorithms from ECHO. For example, for *Avrora*, the exhaustive analysis takes 4.8h, and the four incremental algorithms (reset-recompute, reachability-based, *IPA-1* and *IPA-48*) take 52s, 76s, 89ms and 27ms respectively per deletion on average. However, for the slowest case, the four incremental algorithms take 30+mins, 30+mins, 228s and 10.5s respectively. The reason is that there is often a small number of complex array load and store statements in each benchmark, which involve a long chain of dependencies and a large points-to set. These statements produce a large number of edge changes upon their deletions and incur long change propagation paths. However, such special cases are very rare. As we will discuss in Section 3.5.3, for more than 99.9% of the statements, IPA takes under 1s.

Table 3.4: Performance of the incremental algorithms for handling addition. Reprinted with permission from [1].

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

3.5.2 Performance of Incremental Addition

Table 3.4 compares the performance between the exhaustive pointer analysis and the incremental algorithms for insertion. We note that the existing incremental approaches for handling insertion are essentially the same: they follow the same basic procedure as the fixed-point based on-the-fly pointer analysis algorithm (though IPA has small implementation optimizations). We hence do not duplicate the numbers for these approaches because they are mostly the same as IPA. For REVISER, it has similar performance as reported in Table 3.3 to handle incremental additions, which is much slower than IPA.

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

Table 3.5: Statistics of the special cases. M: #incoming neighbours of the root node after deleting a statement. N: #updated points-to sets after the change. Reprinted with permission from [1].

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

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

3.5.3 Analysis of the Special Cases

The results in Tables 3.3 and 3.4 show that handling deletion is much slower than handling insertion, and in a few cases a statement deletion can take over 20s for IPA to analyze. For example, for *Jython*, both *IPA-1* and *IPA-48* takes less than 0.1s to analyze an insertion in the worst case, whereas for deletion they take 480s and 22s, respectively. The reasons for such special cases are twofold. First, the algorithm for handling deletion is inherently more complex than that for insertion. For the slow cases, the in and out degrees of a change impacted node are typically very large and the chain of dependent variables is long, which require checking a large number of incoming neighbours and propagating the deletion changes to a large number of outgoing neighbours for each node. Second, the underlying on-the-fly pointer analysis implementation provided in WALA is optimized for handling insertion but not for deletion, since the incremental insertion algorithm follows the same flow as that of the exhaustive pointer analysis for each statement.

However, such special cases are very rare. Table 3.5 reports the number of special statements in each benchmark of which the root node in the PAG corresponding to the deletion has $M > 1000$ incoming neighbours and of which the deletion requires updating $N > 100$ points-to sets, which typically take IPA much longer to analyze than by the rest of the statments. Overall, the percentage of such special statements over all statements in the program is very small. The maximum percentage of such cases in the benchmarks is only 3% (in *Fop*), and the percentage for the majority of the benchmarks is under 1‰. And for $M * N > 100K$, the maximum percentage of such cases is only 1‰. In other words, out of every one hundred statements only one of them can affect a node with more than 1K incoming neighbours, and only three statements can affect more than 100 points-to sets. And only one of every one thousand statements can affect both affect more than 1K incoming neighbours and more than 100 points-to sets, which are expensive to analyze. The majority of the worst cases in each benchmark belong to this small category. For the rest 99.9% cases, they typically take IPA several milliseconds or at most 1s to process.

3.5.4 Application on Interactive Race Detection

We have also evaluated IPA for improving the performance of interactive race detection implemented in ECHO [27]. The ECHO tool relies on incremental pointer analysis to detect data races incrementally, in which pointer analysis is used to determine the thread shared objects and synchronization locks. We run both the default ECHO (which uses a hybrid of reset-recompute and the reachability-based algorithm for incremental pointer analysis) and the ECHO with IPA configured with a pool of 48 threads (*IPA-48*) for all the 13 multithreaded applications in DaCapo-9.12 (except *fop* which is single-threaded).

The results are reported in Table 3.6. Column 2 (*Full*) reports the time taken by the one-shot whole program race detection (*i.e.*, with no incremental analysis). The whole-program race detection takes from half an hour to seven hours to analyze a benchmark (*e.g.*, 6.9h for *Avrora* and *Batik*) and 2.8h on average. Columns 3-4 report the time taken by the original ECHO for a

Table 3.6: Performance of incremental race detection. #Slow: statements that take the tool ≥1s. Reprinted with permission from [1].

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

statement insertion and deletion on average, respectively. The time for ECHO includes that taken by the incremental algorithms for updating the PAG and for detecting races per change. With incremental analysis, ECHO is much faster than the whole-program race detection. ECHO takes under 0.3s to process a statement insertion and 27s to process a statement deletion on average, two to three orders of magnitude faster than the whole-program race detection. Column 5 (*#Slow(‰)*) reports the number and the per mille of the slow statements, *i.e.*, those statements that take the tool at least 1s to analyze. For all benchmarks, 1-10‰ of the statements (4‰ on average) require more than 1s by ECHO.

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

(0.2‰ on average), which reduces the number of slow statements by the original ECHO by 10X-100X. In other words, with IPA, for 99.9% of the cases ECHO now takes less than 1s to analyze an incremental statement change.

3.5.5 Discussion on Memory Usage

Because IPA stores the intermediate graphs in memory, for large programs it may require a large memory to run continuously. In particular, in our current implementation we do not remove the PAG and call graph nodes even if they have no incoming and outgoing edges after a statement deletion (in order to improve performance in case the statement is added back later). In our experiments, the largest memory consumption we observed was around 140GB. This may be too excessive for laptops running IDEs, where memory resource is limited. To reduce memory usage, we can periodically clean those graph nodes. Alternatively, we can offload IPA and the client analyses (*e.g.*, data race detection) to remote servers [3] and integrate with a language-independent IDE through the Language Server Protocol (LSP) [167]. We plan to implement IPA in the WALA IDE [168] (which supports LSP) in future work.

3.6 Summary

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

4. SHARP: INCREMENTAL CONTEXT-SENSITIVE POINTER ANALYSIS *

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

4.1 Introduction

The previous chapter introduces a promising algorithm that computes the points-to results *incrementally* without losing precision. The key is to observe a change-locality property of Andersen's analysis [40]: the updates of points-to results upon a code change can be determined by the local neighbors of the change in the pointer assignment graph (PAG) [42]. This allows developing a sound incremental pointer analysis that only recomputes the change impact by leveraging the memoized intermediate analysis results, instead of re-running an exhaustive pointer analysis for every code change. Their algorithm has scaled to large complex Java programs with averaged response time under a second (per statement change), making it suitable to be applied in the programming phase to detect sophisticated bugs such as data races.

*Reprinted with permission from "SHARP: Fast Incremental Context-Sensitive Pointer Analysis for Java" by Bozhen Liu and Jeff Huang. Proceedings of the ACM on Programming Languages, Volume 6, Issue OOPSLA1, Article No.: 88, pp 1–28, Copyright 2022 by Bozhen Liu.

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

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

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

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

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

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

Figure 4.1: A Java example with four statement changes: ①-④. Reprinted with permission from [2].

However, their incremental algorithm only applies to context-insensitive pointer analysis, and hence can be imprecise for many applications. More importantly, it is unclear how to extend their algorithm with context-sensitivity. To illustrate the problem, consider the example in Figure 4.1 involving two threads. Before the four highlighted changes (delete statement ①② and add statement ③④), if we run an incremental static data race detector, D4 [3], it reports five warnings on field f:

- lines (14,14);
- lines (14,15);
- lines (14,16);
- lines (15,15);
- lines (15,16).

Unfortunately, these are all false positives because D4 is based on context-insensitive pointer analysis, which fails to distinguish the two instances of f from different call sites: o_2 from $S1$ and o_4 from $S2$. The strength of D4 is that when the developer pushes code commits it can instantly detect new races or invalidate old warnings. After the four code changes ①-④, D4 still reports one false positive on lines (14,14).

It would be highly desirable for developers to have a fast debugging tool that works incrementally like D4 but with much higher precision, since false positives can significantly reduce the debugging efficiency. To achieve a better precision, *k-CFA* [126] adopts a *k-call-site-sensitive* algorithm with a context-sensitive heap during the on-the-fly call-graph (CG) construction [42]. However, *k-CFA* is known to be unscalable [81, 150]; this applies to other context-sensitive algorithms such as *k-object-sensitive* [49] (denoted *k-obj*) and *k-type-sensitive* [46] (denoted *k-type*).

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

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

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

1. The first contribution is a new algorithm that enhances the existing incremental algorithms [1, 3] with context-sensitivity of *k-CFA* and *k-obj*. In particular, we discover the redundant computation in the fixed-point-based pointer analysis when a deleted new statement or method invocation destroys a sequence of method calls and contexts. We utilize two common properties in both contexts: *inheritance* and *uniqueness*, and leverage them to identify and remove all invalid program elements in advance to avoid useless propagation of changes generated by iteratively identified invalid method calls.

2. The second contribution is a parallel algorithm that further scales the proposed incremental algorithm by leveraging a third property, *convergence*. Besides, we study the incremental impact on the PAG from real-world code commits, and discuss various parallel scenarios from the perspective of efficiency, redundancy and conflict. Instead of pre-ordering multiple changed statements [94, 97], our parallel algorithm minimizes conflict while avoiding redundant computation.

3. The third contribution is an extensive evaluation of SHARP on real-world code commits. We implemented an end-to-end incremental *k-CFA* and *k-obj* algorithm, and conducted an empirical evaluation on a collection of popular and actively maintained Java projects on GitHub, *i.e.*, Hbase, Lucene, Yarn, Zookeeper. The evaluation shows similar efficiency and scalability as the state-of-the-art context-insensitive incremental algorithm [1, 3]: SHARP requires 31s on average to handle all the statement changes from a real-world git commit; our parallel algorithm achieves on average 1.3x speedup over our incremental algorithm on an eight-core machine.

4.2 Recall IPA

Section 3.2.1 introduces IPA in the form of algorithms, which we reform into the form of inference rules to be consistent with the rules of SHARP. This section illustrates the inference rules of IPA.

IPA takes statement changes as input and efficiently updates the change-affected part in the PAG and CG. For each deleted statement, IPA extracts the pointer assignment edge(s) according to Table 2.1 and follow the inference rules shown in Figure 4.2 to handle each deleted pointer assignment edge. The procedure of resolving incremental statement additions is similar but simpler, so this section focuses on the illustration of how to handle incremental deletions.

DELETEPAGEDGE takes a deleted pointer assignment edge e_{pag} as input, remove this edge from the PAG, and calls different rules according to the type of e_{pag}. The called rules (*i.e.*, DELETEPOINTSTO, DELETELOAD, DELETESTORE and DELETEINVOKE) take e_{pag} and a set of points-to

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

$$\begin{cases} \textbf{\textit{if}}\ e_{pag} = o \to x \vee y \to x : \textsc{DeletePointsTo}(e_{pag}, pts(y)) \\ \textbf{\textit{if}}\ e_{pag} = y \rightsquigarrow x.f \vee y \rightsquigarrow x.* : \textsc{DeleteLoad}(e_{pag}, pts(y)) \\ \textbf{\textit{if}}\ e_{pag} = y.f \rightsquigarrow x \vee y.f \rightsquigarrow x : \textsc{DeleteStore}(e_{pag}, pts(x)) \\ \textbf{\textit{if}}\ e_{pag} = y \overset{invoke\ g()}{\rightsquigarrow} : \textsc{DeleteInvoke}(e_{pag}, pts(y)) \end{cases}$$

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

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

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

$$\textsc{DeleteInvoke}\ \frac{y \overset{invoke\ g()}{\rightsquigarrow} \qquad \Delta}{\textbf{\textit{if}}\ deleted, E_{cg} \setminus \{m' \rightarrowtail m\}}$$

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

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

$$\textsc{Propagate}\ \frac{pts(y) = pts(y) \setminus \Delta \qquad\qquad \Delta \neq \emptyset}{\begin{gathered} \forall y \to v \in E_{pag} : \textsc{Check}(v, \Delta) \\ \forall e_{pag} = y.f \rightsquigarrow w \in E_{pag} : \textsc{DeleteLoad}(e_{pag}, \Delta) \\ \forall e_{pag} = w \rightsquigarrow y.f \in E_{pag} : \textsc{DeleteStore}(e_{pag}, \Delta) \\ \forall e_{pag} = y \overset{invoke\ g()}{\rightsquigarrow} \in E_{pag} : \textsc{DeleteInvoke}(e_{pag}, \Delta) \end{gathered}}$$

Figure 4.2: IPA: DELETEPAGEDGE: The inference rules of handling a deleted pointer assignment or dynamic edge in the PAG = (N_{pag}, E_{pag}) and CG = (N_{cg}, E_{cg}).

set changes Δ as input, where Δ is initialized by this rule.

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

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

DELETELOAD is called on a deleted dynamic edge generated by a LOAD FIELD or LOAD ARRAY statement: the dynamic edge (*i.e.*, $y.f \rightsquigarrow x$ for FIELD or $y.* \rightsquigarrow x$ for ARRAY) and its derived pointer assignment edges (*i.e.*, $o.f \rightarrow x$ for FIELD or $o.* \rightarrow x$ for ARRAY) are deleted, where $o \in \Delta$ is removed from $pts(y)$. If this rule is called by PROPAGATE, the dynamic edge $y \rightsquigarrow x$ is kept in the PAG. However, $o \in \Delta$ is no longer valid in $pts(y)$ so that the derived pointer assignment edges should be deleted.

Similarly, **DELETESTORE** is applied on a deleted dynamic edge from STORE FIELD or STORE ARRAY statements, or propagating a change Δ for such a dynamic edge from y to x.

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

4.3 Scope and Limitations of IPA

Correct context is crucial while building PAG and CG for *k-CFA* and *k-obj*, which cannot be randomly decided for a statement change. Except for context, there exist redundant computations when applying IPA to context-sensitive algorithms. In this section, we firstly explain how IPA works with an example, then illustrate the scope and limitation of IPA.

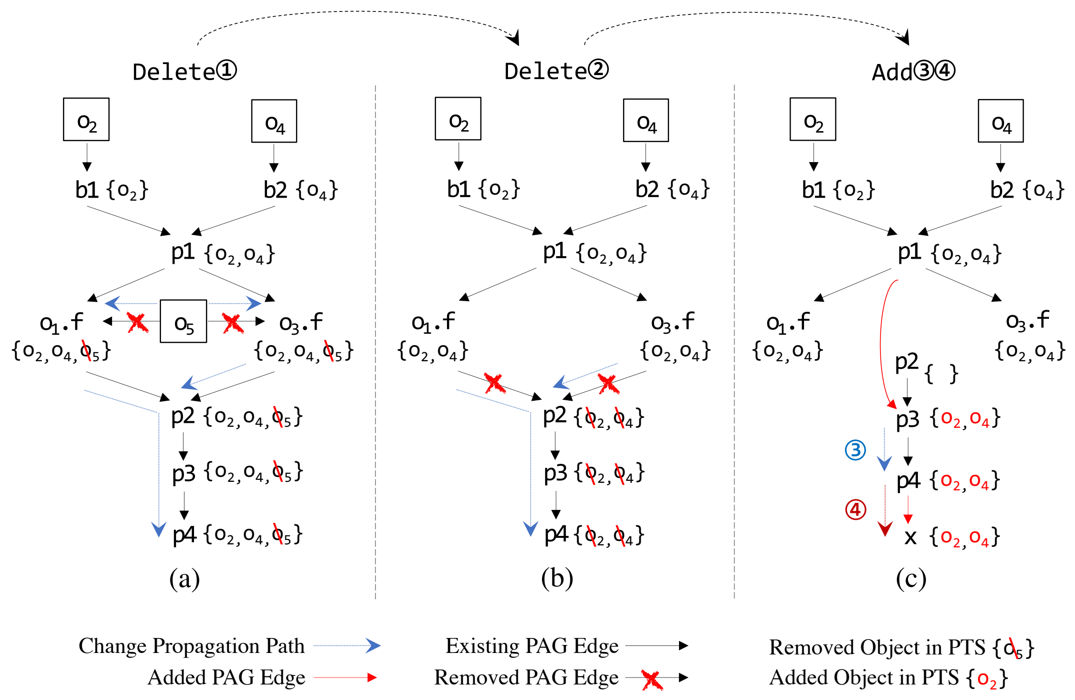


Figure 4.3: The context-insensitive PAG changes for code changes ①②③④ in Figure 4.1. Reprinted with permission from [2].

4.3.1 Illustration of IPA

We use the statement changes ①-④ in the code shown by Figure 4.1 to illustrate the rules of IPA. As depicted in Figure 4.3, IPA handles statement deletions ①② at first, and then additions ③④.

- **Delete ① `f = new B():`** IPA extracts the pointer assignment edges, calls DELETEPAGEDGE and removes edges $o_5 \to o_1.f$ and $o_5 \to o_3.f$. Then DELETEPOINTSTO is called with the initialized change $\Delta = \{o_5\}$. CHECK and PROPAGATE are performed on the removed edges, sequentially.

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

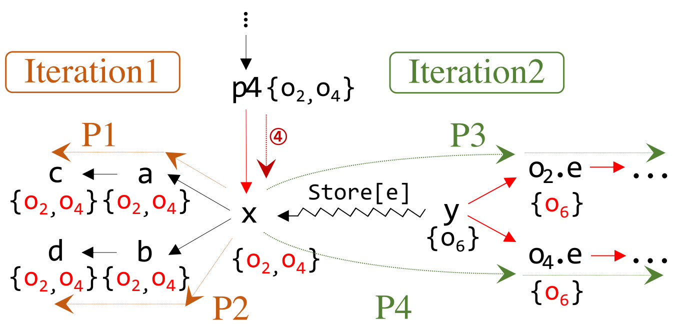


Figure 4.4: The parallel propagation after adding statement `x = p4`. Reprinted with permission from [2].

applied to $p2$, $p3$ and $p4$, successively.

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

- **Delete ② `m2(f)`:** According to DELETEINVOKE, IPA deletes the edges $o_1.f \rightarrow p2$ and $o_3.f \rightarrow p2$ which represent the points-to relation between the actual and formal parameters for the removed method call to $m2()$. Both deleted edges have the change initialized to $\Delta = \{o_2, o_4\}$. Then IPA runs CHECK on $p2$ and finds out it has no incoming pointer assignment edges. Hence, the change $\Delta = \{o_2, o_4\}$ is confirmed and removed from $pts(p2)$, and then propagated to $p3$. Afterwards, the CHECK and PROPAGATE are applied on $p3$ and $p4$ in order.

- **Add ③ `m3(p1)`:** An edge $p1 \rightarrow p3$ is added in Figure 4.3(c) for the parameter of method $m3()$. This is based on the Andersen's algorithm for INVOKE statements in Table 2.1. Simpler than handling a deletion, IPA checks whether $pts(p3)$ contains the change $\Delta = pts(p1) = \{o_2, o_4\}$, adds Δ to $pts(p3)$ and propagates Δ to $p4$.

- **Add ④ `x = p4`:** IPA adds the pointer assignment edge $p4 \rightarrow x$ by following the rule for ASSIGN statement and propagates the change $\Delta = pts(p4) = \{o_2, o_4\}$ to x .

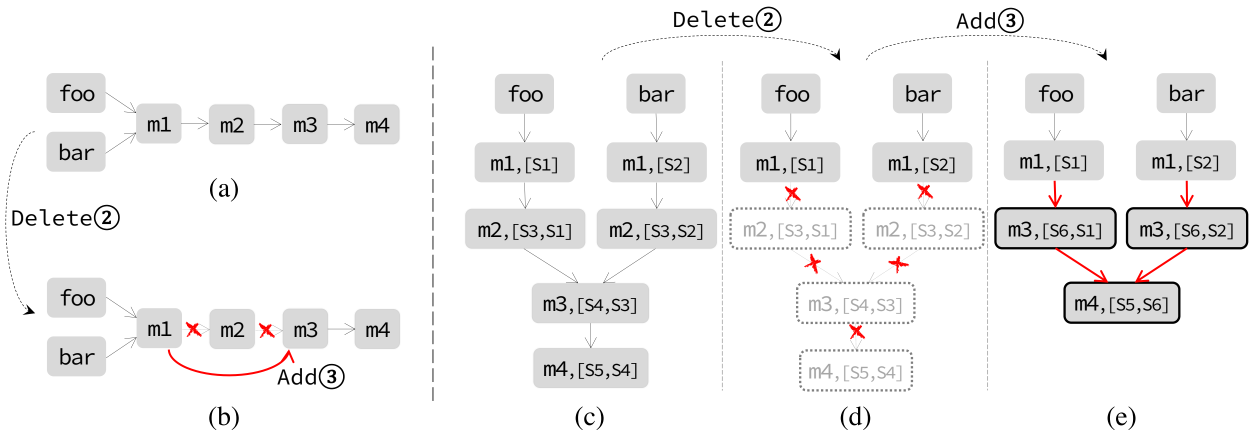


Figure 4.5: The CG after each statement change in Figure 4.1. (a)(b) Context-insensitive CG. (c)-(e) 2-CFA CG. Reprinted with permission from [2].

We use the PAG in Figure 4.4 to explain the workflow of PIPA. After adding statement ④, PIPA confirms the change $\Delta = \{o_2, o_4\}$ in $pts(x)$ through CHECK and starts to propagate Δ to all the outgoing neighbours of x in parallel.

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

4.3.2 Scope and Limitations

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

For example, replacing ② by ③ is simple for the context-insensitive CG as shown in Figure 4.5(a)(b) according to IPA: for the deletion of ②, the call edges $m1 \rightarrowtail m2 \rightarrowtail m3$ are removed.

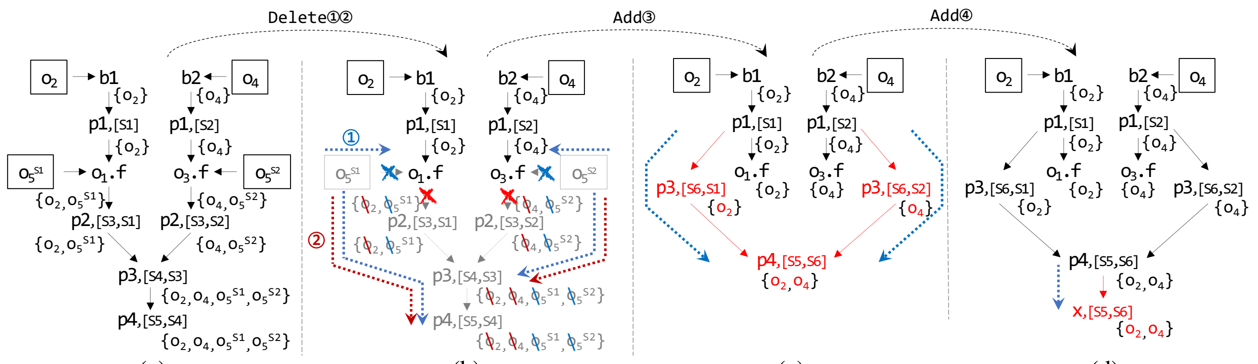


Figure 4.6: The 2-CFA PAG after each statement change in Figure 4.1. Reprinted with permission from [2].

Now $m3$ has no caller, so nodes $m3$, $m4$ and their connected edge become invalid which should be removed. Nevertheless, IPA keeps them in case that a future change (adding ③) will add them back. So do the pointers and edges created by $m3$ and $m4$ in the PAG. This *reuse trick* is only valid in context-insensitive algorithm, because multiple calls share the same target method. In summary, only one node is removed from CG for a method call replacement.

However, for 2-CFA, deleting ② causes four nodes removed from its CG as shown in Figure 4.5(d), and adding ③ creates three new nodes in Figure 4.5(e). Here, the reuse trick cannot be adopted, because the deleted contexts can never be retrieved by later changes, unless the previous deletion of ② is withdrawn. In total, seven nodes are changed, not to mention the consequent update in the 2-CFA PAG.

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

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

Consider the deletion ② in Figure 4.5(c)(d), which makes $S3$ an invalid call site. As a result, methods $\langle m2, [S3, S1]\rangle$ and $\langle m2, [S3, S2]\rangle$ are invalid and removed. Then their parameter constraints are solved as shown in Figure 4.6(b): IPA resets $pts(\langle p2, [S3, S1]\rangle)$ and $pts(\langle p2, [S3, S2]\rangle)$ and propagates the changes $\{o_2, {o_5}^{S1}\}$ and $\{o_4, {o_5}^{S2}\}$, respectively. $pts(\langle p3, [S4, S3]\rangle)$ and $pts(\langle p4, [S5, S4]\rangle)$ are updated twice, each for one change as shown in Figure 4.6(b) (the first time is updated by the blue path, and the second time is updated by the maroon path). Later, IPA deletes the statements enclosed in the two removed methods, which removes CG edges $\langle m2, [S3, S1]\rangle \rightarrowtail \langle m3, [S4, S3]\rangle$ and $\langle m2, [S3, S2]\rangle \rightarrowtail \langle m3, [S4, S3]\rangle$. Now, $S4$ is discovered to be invalid, which indicates the previous change propagations are redundant. A more efficient way is to simply empty their points-to sets instead of running the CHECK and PROPAGATE rules.

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

More Parallelization The parallel algorithm of PIPA leaves plenty of room for improvement. For example, can the changes be propagated along all the paths in Figure 4.4, *e.g.*, *P1* - *P4*, in parallel? Can multiple statement changes be handled in parallel? If possible, can the multiple statements come from one method or different methods? Is there any conflict or redundancy while updating points-to sets during the propagation? Moreover, is it possible to add more parallelization for incremental context-sensitive algorithms?

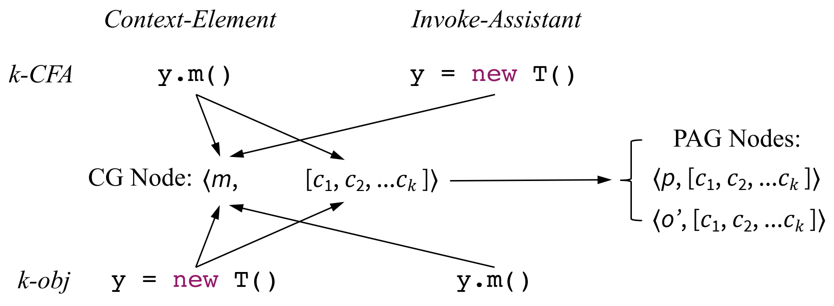
4.4 SHARP

We now introduce, SHARP, our new incremental algorithm designed for *k-CFA* and *k-obj*. Here, we focus on how to efficiently handle statement deletions, which is challenging for incremental analyses [1, 27, 97]. Handling additions is then straightforward.

Definitions A precise pointer analysis requires iterative computation between PAG and CG, in order to achieve a precise result. This introduces more computation workload when considering

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

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



(b)

Figure 4.7: How NEW and INVOKE determine the CG and PAG for *k-CFA* and *k-obj*, respectively. (a) An example code. (b) The Context-Element and Invoke-Assistant Statements for *k-CFA* and *k-obj*, and how they affect the CG and PAG. Reprinted with permission from [2].

contexts. For Java programs, there are three facts that determine a context-sensitive CG node: (1) an INVOKE statement, (2) the abstract heap object that the base variable of the invocation may point to and (3) context.

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

A specific type of statement defines the context elements for a particular context-sensitive algorithm as shown in Figure 4.7: an INVOKE statement defines a call site S as the context element of *k-CFA*, and a NEW statement defines an abstract heap allocation site O as the one of *k-obj*. We define this specific type of statement the *Context-Element Statement* (denoted SCTX).

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

In this paper, we overload the $\in$ operator to more than set operations, for example: $s \in \{\text{NEW}\}$ (statement s is a NEW statement), SCTX $\in m$ (statement SCTX is from its enclosing method m),

$c \in C$ (context element c is an element of context C).

Definition 4.4.1 (Context-Element Statement)

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

Definition 4.4.2 (Invoke-Assistant Statement)

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

To be specific, as shown in Figure 4.7(b), SCTX refers to INVOKE statements and SINV refers to NEW statements for *k-CFA*. While it is the opposite for *k-obj*: SCTX refers to NEW and SINV refers to INVOKE. No matter which context is selected in pointer analysis, both NEW and INVOKE statements determine a CG node, which further defines a set of PAG nodes. Hence, we need to carefully handle the deletion of NEW and INVOKE statements if we want to efficiently identify invalid graph elements.

Properties We leverage two properties in context-sensitive CG:

- INHERITANCE: a caller node $\langle m', [c_1, c_2, ...c_k]\rangle$ must share the first $(k - 1)$ context elements (*i.e.*, call site s in *k-CFA* and object allocation site o in *k-obj*) with its callee node $\langle m, [c, c_1, ...c_{k-1}]\rangle$.

- UNIQUENESS: one SCTX describes its unique context element c, and defines a set of unique contexts C where $c \in C$. C then defines a unique CG node $\langle m, C\rangle$ together with method m, and a unique set of pointer nodes $\langle y, C\rangle$, object nodes $\langle o, C\rangle$ and edges in PAG.

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

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

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

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

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

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

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

Figure 4.8: SHARP: PREDEL: The inference rules to precompute the impact on CG = (N_{cg}, E_{cg}) after deleting SCTX and SINV from their enclosing method m' with context C'. Reprinted with permission from [2].

4.4.1 Precompute for Deletion

The inference rules to precompute the impact on CG from a deleted SCTX and SINV are shown in Figure 4.8: N^-_{cg} and E^-_{cg} represent two sets of CG nodes and edges that are no longer valid due to the deletion, and N^*_{cg} represents all points-to constraints originated from a statement should be deleted from its enclosing CG node. We overload the $\Rightarrow$ operator to more operations: $s \Rightarrow c$ means s determines a context element c, and $e_{cg} \Rightarrow s$ means e_{cg} is introduced by statement s. The red square brackets ([]) means that the premises or conclusions are limited to the specific context-sensitive algorithm indicated by the brackets. Next, we illustrate how the rules work for *k-CFA* and *k-obj*, respectively.

4.4.1.1 k-CFA

PREDEL identifies the type of input statement s and selects the correct precompute rules (*i.e.*, SCTX-RULE or SINV-RULE) for different contexts (*i.e.*, *k-CFA* or *k-obj*).

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

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

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

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

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

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

4.4.1.2 k-Obj

This section introduces the differences when applying the rules in Figure 4.8 to *k-obj*.

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

SINV-Rule is called to handle a deleted INVOKE statement s in method m', where s is of shape `x = y.g(..., `a_i`, ...)`. There are two differences from the rule applied to *k-CFA*: (1) s and

$$\textsc{DeleteStmt}\ \frac{s}{\begin{cases} \textbf{\textit{if}}\, s \in \{\textsc{Assign}, \textsc{Load}, \textsc{Store}\} : \text{PAGO}\textsc{nly}(s) \\ \textbf{\textit{if}}\, s \in \{\textsc{New}, \textsc{Invoke}\} : \text{PAGA}\textsc{nd}\text{CG}(s) \end{cases}}$$

$$\text{PAGO}\textsc{nly}\ \frac{s \in \{\textsc{Assign}, \textsc{Load}, \textsc{Store}\} \qquad C \in context(m) \qquad \mathcal{E}_{pag} = extract(s, C)}{\forall e_{pag} \in \mathcal{E}_{pag} : \qquad \textsc{Delete}\text{PAGE}\textsc{dge}(e_{pag})}$$

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

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

$$\textsc{PropagateOrReset}\ \frac{e^{-}_{pag} = v \rightarrow y \vee v \rightsquigarrow y \in \mathcal{E}^{-}_{pag}}{\begin{array}{c} \forall e_{pag} = y \rightarrow a \vee y \rightsquigarrow a \vee v \rightarrow a \vee v \rightsquigarrow a : \\ \textbf{\textit{if}}\ e_{pag} \notin \mathcal{E}^{-}_{pag} \wedge e_{pag} \in E_{pag} : \textsc{Delete}\text{PAGE}\textsc{dge}(e_{pag}) \\ N_{pag} = N_{pag} \setminus \{v, y\} \end{array}}$$

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

Figure 4.9: SHARP: DELETESTMT: The inference rules for a deleted statement s in method m under *k-CFA* and *k-obj*. Reprinted with permission from [2].

s' are the same INVOKE statement which introduce the dynamic edge; (2) we delete the introduced call edge e_{cg} since the call relation is no longer available after this deletion.

CHECKVALIDITY has no difference comparing with the one applying for *k-CFA*. Afterwards, we call the rules **CHECKNEWSCTX** and **CHECKNEWSINV**.

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

CHECKNEWSINV finds the invalid calls introduced by INVOKE statements from N^-_{cg} and SINV-RULE should be applied to them.

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

4.4.2 Formulations for Deletion

Figure 4.9 shows the inference rules to incrementally update the PAG and CG for a deleted statement s when considering contexts.

DELETESTMT takes an deleted statement s as input, and matches the type of s with the two rules PAGONLY and PAGANDCG.

PAGONLY handles a deleted statement of type ASSIGN, LOAD and STORE. Because they only introduce changes in the PAG before we perform any change propagation. *extract(s, C)* is a function to extract pointer assignment edges for a statement s under context C according to Table 2.1.

PAGANDCG is designed to handle deletions of type NEW and INVOKE by utilizing the impact on CG precomputed by the rules in Figure 4.8. Note that we firstly apply DELETECGNODE to each invalid CG node $n^-_{cg} \in N^-_{cg}$, then we handle the constraint deletions from $[s^*, n^*_{cg}] \in N^*_{cg}$ by DELETECONSTRAINT.

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

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

$pts(y)$. Finally, we simply remove v and y from PAG.

DELETECONSTRAINT extracts the pointer assignment edges introduced by statement s^* from method m under context C. All the extracted pointer assignment edges $\mathcal{E}^*_{pag}$ should be removed, and we apply the rules in Table 2.1to each e^*_{pag} according to its type.

4.4.3 Illustration of SHARP

We use the two deletions ① and ② in Figure 4.1 to illustrate the inference rules in Figure 4.8 and 4.9 when applying them for 2-CFA.

- **Delete ① `f = new B()`:** Because the deleted statement is of type NEW with the context 2-CFA, DELETESTMT concludes that PAGANDCG should be used and calls PREDEL. Then PREDEL determines SINV-RULE is the correct precompute rule.

  SINV-RULE firstly collects all method calls invoked by the object field variable f under its possible contexts, *i.e.*, calls using the base objects $\langle o5, [S1]\rangle$ (from $pts(o1.f)$) and $\langle o5, [S2]\rangle$ (from $pts(o3.f)$) as shown in Figure 4.6(a). Since there is no such call in the PAG, the concluded N^-_{cg} and E^-_{cg} are empty, which means no need to use the rules of CHECKNEWSCTX and CHECKNEWSINV and hence no updated in the CG. We add two mappings to N^*_{cg}: the deleted statement with its two enclosing CG nodes, *i.e.*, [①, $\langle m1, [S1]\rangle$] and [①, $\langle m1, [S2]\rangle$].

  Continuing with PAGANDCG, we skip DELETECGNODE because of the empty return value N^-_{cg}. Then DELETECONSTRAINT is applied to N^*_{cg}: we extract the two pointer assignment edges, $\langle o5, [S1]\rangle \rightarrow \langle o1.f\rangle$ and $\langle o5, [S2]\rangle \rightarrow \langle o3.f\rangle$, and perform DeletePAGEdge on the extracted edges as shown in Table 2.1.

- **Delete ② `m2(f)`:** SCTX-RULE is used here for an deleted INVOKE statement. As shown in Figure 4.1, ② determines the context element $S3$ in its enclosing method $m1$ (*i.e.*, $s \Rightarrow S3$ and $s \in m1$). Hence, we can conclude that all the CG nodes with $S3$ as its context element are invalid, *i.e.*, $\langle m2, [S3, S1]\rangle$, $\langle m2, [S3, S2]\rangle$ and $\langle m3, [S4, S3]\rangle$ should be included in

N_{cg}^{-}. As shown by Figure 4.5(d), all the CG edges that point to the nodes in N_{cg}^{-} are also invalid, which are excluded from CG and included into E_{cg}^{-}. To guarantee the soundness, $[②, \langle m1, [S1]\rangle]$ and $[②, \langle m1, [S2]\rangle]$ are added to N_{cg}^{*}.

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

Then CHECKNEWSCTX and CHECKNEWSINV are used on E_{cg}^{-} and N_{cg}^{-}, respectively. CHECKNEWSCTX collects all INVOKE statements in the callee node of a call edge in E_{cg}^{-} (*i.e.*, here the callee nodes are $\langle m2, [S3, S1]\rangle$, $\langle m2, [S3, S2]\rangle$, $\langle m3, [S4, S3]\rangle$ and $\langle m4, [S5, S4]\rangle$, which are the same as N_{cg}^{-}) and applies SCTX-RULE to them. CHECKNEWSINV collects all New statements enclosed in the nodes from N_{cg}^{-} and applies SINV-RULE to them. After this round of running SCTX-RULE and SINV-RULE, we found out there is no newly concluded E_{cg}^{-} and N_{cg}^{-}, and exit PREDEL.

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

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

4.4.4 Handling Addition

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

Algorithm 7: The End-to-End SHARP.

```
Input      : $commit$ - a new git commit hash.
GlobalState: Prog - the analyzed program,
             CG - the call graph,
             PAG - the pointer assignment graph,
             $prevCommit$ - the first parent of $commit$.
1  GitCheckOut($commit$) // run git checkout to obtain this commit
2  Build(Prog) // rebuild the program with the new code change from
       commit
3  $D \leftarrow \emptyset$, $A \leftarrow \emptyset$ // initialize
4  $\langle java^-, java^+, java^* \rangle \leftarrow$ GitDiff($prevCommit$, $commit$)
5  foreach $f \in java^*$ do
6      $\{method^*\} \leftarrow$ GetEnclosingMethods($f$)
7      foreach $m^* \in \{method^*\}$ do
8          $\langle \{stmt^-\}, \{stmt^+\} \rangle \leftarrow$ CompareIRDiff($m^*$) // compare new and old IRs
9          $D = D \cup \{stmt^-\}$
10         $A = A \cup \{stmt^+\}$
11     end
12 end
13 foreach $s^- \in D$ do
14     DELETESTMT($s^-$) // handle deletions according to Figure 4.9
15 end
16 foreach $s^+ \in A$ do
17     ADDSTMT($s^+$) // handle additions according to Table 2.1
18 end
19 $prevCommit \leftarrow commit$
```

4.4.5 The End-to-End Algorithm

This section introduces the end-to-end algorithm of SHARP after running the initial pointer analysis for the whole program. Algorithm 7 takes each code commit as an input to compute the impact on the PAG and CG from the code changes.

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

For each modified java file f, we retrieve a set of its enclosing methods $\{method^*\}$. For each such method m^*, *CompareIRDiff* computes the new IR for the new commit and compares with its old IR from $prevCommit$ to obtain the added and deleted statements. Next, we iterate each delete statement in D and apply the rule DELETESTMT to them. Finally, we handle the additions by following the Andersen's algorithm in Table 2.1 with context selections.

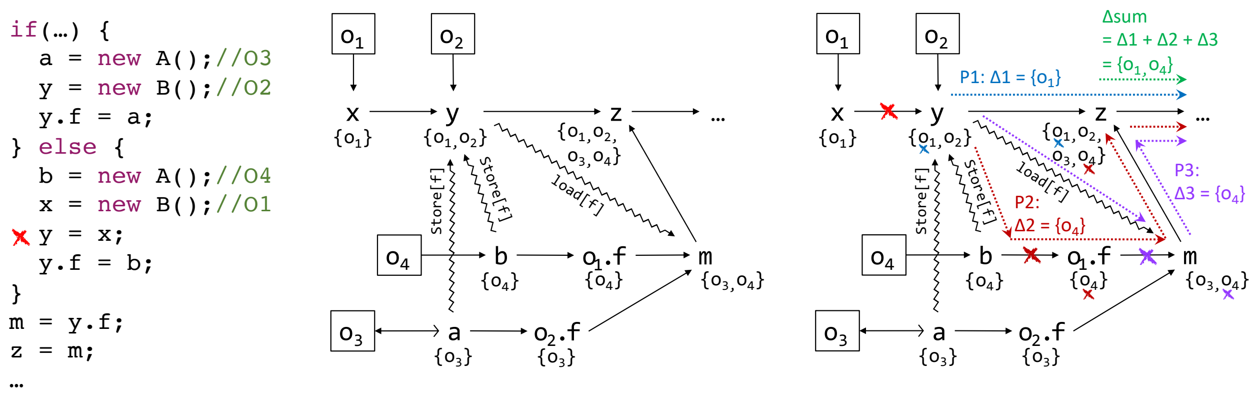
4.5 Parallel Algorithms

4.5.1 Context Level

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

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

DELETECGNODE and its conclusion PROPAGATEORRESET can be executed in parallel for different invalid CG nodes introduced by the same NEW or INVOKE statement.



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

(a) (b) (c)

Figure 4.10: An example to explain the intraprocedural parallelization. (a) Delete y = x from the example code. (b) The PAG before the deletion. (c) The three change propagation paths after the deletion. Reprinted with permission from [2].

4.5.2 Code Change Level

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

4.5.2.1 Intraprocedural Changes.

There are two parallel scenarios for the statement changes within a method: propagate points-to set changes in parallel for one changed statement (Section 4.5.2.1) and for multiple changed statements (Section 4.5.2.1).

One Changed Statement Consider the example in Figure 4.10. Suppose we delete statement y = x from the code in Figure 4.10(a), which leads to the deletion of edge $x \rightarrow y$. As shown in Figure 4.10(c), we run our incremental algorithm that deletes the edge $x \rightarrow y$ with $\Delta = pts(x) = \{o_1\}$ and removes Δ from $pts(y)$. Now, three tasks of running the PROPAGATE rule for y will be performed sequentially in the following order:

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

Task 2: Propagation for the dynamic edge $b \overset{store[f]}{\rightsquigarrow} y$ in Figure 4.10 (*i.e.*, $b \rightsquigarrow y.f$) along the path $P2$ (the maroon path): according to DELETESTORE, the edge $b \rightarrow o_1.f$ is extracted for $\Delta 1 = pts(y) = \{o_1\}$ and should be deleted, which initializes the change $\Delta 2 = pts(b) = \{o_4\}$. Then CHECK is applied to see whether any existing incoming neighbor of $o_1.f$ has o_4 in its point-to set. Since there is no such node, the change $\Delta 2$ is confirmed and removed from $pts(o_1.f)$. Later, $\Delta 2$ is propagated to m, z and its reachable nodes, consecutively.

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

Task 3: Propagation for the dynamic edge $y \overset{load[f]}{\rightsquigarrow} m$ in Figure 4.10 (*i.e.*, $y.f \rightsquigarrow m$) along the path $P3$ (the purple path): similar to Task 2, DELETELOAD extracts edge $o_1.f \rightarrow m$ with the initialized change $\Delta 3 = pts(o_1.f) = \{o_4\}$. Then $\Delta 3 = \{o_4\}$ is confirmed by CHECK, which is removed from $pts(m)$ and propagated to z and its reachable nodes.

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

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

Case (i) $P_{sum} = \emptyset$,

Case (ii) $P_{sum} \neq \emptyset \ \wedge \ \Delta_{sum} = \Delta_i \neq \emptyset \ \wedge \ \Delta_j = \emptyset$ ($\forall j \in [1, n]$ but $j \neq i$),

Case (iii) $P_{sum} \neq \emptyset \ \wedge \ \Delta_{sum} = \bigcup_{i=1}^{k} \Delta_i \ \wedge \ \Delta_i \neq \emptyset \ (1 < k \leq n)$.

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

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

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

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

Continue with the previous example. If we run the three tasks in parallel, Figure 4.10(c) explains for case (iii):

- $P1$ and $P2$ illustrate Situation 1: $\Delta 1 \bigcap \Delta 2 = \{o_1\} \bigcap \{o_4\} = \emptyset$, no matter which path propagates its change first, the finalized change in $pts(z)$ is always the same.

- $P2$ and $P3$ illustrate Situation 2: $\Delta 2 \bigcap \Delta 3 = \{o_4\} \bigcap \{o_4\} = \{o_4\} \neq \emptyset$, both paths propagates the same change to and finalized at m. For the propagation after m, we have $\delta = \{o_4\}$ and Δ_2^{rest} (if $P3$ is executed firstly) $=$ Δ_3^{rest} (if $P2$ is executed firstly) $= \emptyset$.

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

Multiple Statement Changes We use an SSA-based IR [43] for pointer analysis, where each variable in a method are represented by a unique value number (*e.g.*, v1, v2, v3). After statement changes, IR will be recomputed by assigning new value numbers to variables in the new code. Therefore, a value number represents different variables in the source code before and after the change, which requires a complex procedure to match the new and old values for a variable.

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

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

4.5.2.2 Interprocedural Changes.

This is the most common scenario in our studied code commits: developers change several statements in different methods. According to the end-to-end incremental algorithm [1, 3], it separates the code changes into two sets: statement deletions and additions. They firstly solve all the deletions and then the additions. Here, we follow the same procedure.

One Changed Statement in Each Method Assume there is only one changed statement for each method, and all the changes from different methods are either deletions or additions. Similar to Section 4.5.2.1, there are three cases for a change Δ_i^m along its propagation path P_i^m ($i \in [1, n]$) originated from a changed statement in method $m \in \mathbf{M}$, where $\mathbf{M}$ is a set of different methods with the same type of statement changes. Here also exists a sequence of nodes repetitively updated by some paths with change Δ_{share} along the common path P_{share}:

Case (i) $P_{share} = \emptyset$,

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

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

The three cases are equivalent to the ones in Section 4.5.2.1, so the same proof can be easily adapted to the three cases here to conclude that: the computation of changed statements (either deletion or addition) from multiple methods (one statement from one method) can be parallel without conflict.

Multiple Changed Statements in Each Method We assume multiple methods have changes, and each method includes several statement changes that are either deletions or additions. As we discussed in Section 4.5.2.1, there exist useless computations if we handle multiple statement changes of one method in parallel. Thus, applying a massive parallelization here will introduce more redundancy. Instead, we should handle the statement changes in each method sequentially.

Statement Changes with Both Deletion and Addition We assume multiple methods have changes, each method has only one changed statement which can be deletion or addition. It is obvious that an addition can invalidate a deletion effect, and vice versa. To avoid such invalidation, an optimized schedule of processing statements is compulsory [97], which may or may not gain efficiency over the computation of scheduling. Hence, such a parallel strategy is unwise and we do not adopt it.

4.6 Evaluation

We implemented SHARP in the WALA framework [154] by utilizing its existing k-CFA implementation [43], and implemented k-obj according to the original paper [49]. Our implementation is open-source [†]. We performed an empirical evaluation on four large, real-world Java projects on GitHub with frequent code commits. Our evaluation aims to answer the following research questions:

1. Is SHARP as efficient as IPA?

2. Is our parallel algorithm more efficient than the existing parallel algorithm [1, 3]?

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

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

Table 4.1: GitHub Information (for the 10 git commits). Reprinted with permission from [2]. Reprinted with permission from [2].

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

Table 4.2: The Overview of Sharp and Statistics of Different Pointer Analyses. Reprinted with permission from [2].

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

Benchmarks and Code Commits We selected the Java projects as shown in Table 4.1, all of which are real-world projects that are widely used and have frequent code updates in their GitHub repositories. We perform our evaluation on the most recent 10 git commits (at the date of writing) from the benchmarks' official git repositories in order to evaluate SHARP on the real-world version control system. Here, the code update refers to source code change in *.java* files, and we ignore the git commits that changes non-Java files, *e.g.*, the updates on build files, text files or comments.

Table 4.1 provides the statistic information for the evaluated 10 git commits for each benchmarks. Column 2 shows the total number of added and deleted statements (denoted $+$ and $-$, respectively). Columns 3 and 4 show the total number of deleted NEW and INVOKE statements from the evaluated commits, on which we apply SCTX-Rule and SINV-Rule (Figure 4.8).

Table 4.2 shows the statistics of our evaluated PAGs under different pointer analysis algorithms: Columns 5-7 report the number of pointers, objects and pointer assignment edges in the PAG for each benchmark. All the benchmarks generate very large context-sensitive PAGs, which include from at least 2 millions to at most 1 billions pointer assignment edges.

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

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

To compare the performance of parallelization, we evaluate our parallel algorithm (denoted *SHARP (Max)*) against the existing parallel algorithm [1, 3] (denoted *SHARP (PIPA)*), both of which are running on 1-CFA, 2-CFA, 1-obj and 2-obj with 8 threads.

To make sure the whole program pointer analysis can finish within 12 hours, we exclude certain

Table 4.3: The Performance of Incremental Algorithms on Different Pointer Analyses (Time: ms). Reprinted with permission from [2].

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

libraries from the evaluation, *e.g.*, *java.awt.** and *java.text.**. The experiment is conducted on a Linux machine with Intel Xeon E7-4860 (Westmere-EX), 8-core, 2.26GHz and 1TB memory.

4.6.1 Overview of SHARP

Table 4.2 provides an overview on the performance of SHARP: column 3 shows the full time for running different pointer analysis algorithms (indicated by column 2) for the whole project, and column 4 shows the average end-to-end time to analyze a git commit by SHARP.

In general, SHARP(Max) only requires 18332.64ms on average to finish the incremental analysis for a commit, which is 186.6x (highlight by red) faster than re-running the whole program analysis. Meanwhile, IPA is on average 193.5x faster than re-running the context-insensitive algorithm, which means SHARP(Max) can achieve the similar speedup as IPA does and is as efficient as IPA. The max speedup of SHARP(Max) is 880.6x when analyzing Zookeeper under 2-CFA. This proves the efficiency of SHARP(Max): rather than spending hours to run the whole program pointer analysis, SHARP(Max) provides fast feedback for each commit changes within 19 seconds, which is a super helpful and convenient way for continuous integration apps to utilize the result of pointer analysis.

4.6.2 Performance of Incremental Algorithms

Table 4.3 lists the detailed data in the performance comparison between SHARP and IPA. Columns 4-7 report the performance for each commit: columns 4 and 5 show the average time to compute the sets of added and deleted statements extracted from a commit, columns 6 and 7 report the average and worst time to run our PREDEL rules for the deleted NEW and INVOKE statements in a commit (denoted *PreDel*). Columns 8-12 report the performance for each statement: columns 8-10 show the average time to handle an added statement, a deleted statement, and to run PREDEL for one deleted NEW or INVOKE statement, while columns 11-12 show the worst case addition and deletion time.

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

We can observe that SHARP requires less execution time on *k-obj* than *k-CFA* for most benchmarks in our evaluation, especially the time spend for running the procedure *PreDel*. This is because there are more deleted INVOKE statements (6.51x on average and at most 7.56x) in each git commit as shown in Table 4.1, and the size of PAGs generated by *k-obj* are relatively smaller than the ones by *k-CFA* for the same project.

For some cases, SHARP makes the handling of statement deletions faster than additions due to the full optimizations (*i.e.*, *PreDel* and the rules in Figure 4.9). For example, running SHARP on 2-CFA for Yarn requires 287.48ms on average to handle an addition, and 51.85ms on average (the average of 13.84ms for deletion and 89.85ms for *PreDel*) to solve a deletion.

Table 4.4: The Performance of Parallel Algorithms on Different Pointer Analyses (Time: ms). Reprinted with permission from [2].

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

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

From the perspective of the volume of changes in the PAG and CG, SHARP is still quite efficient to finish such a huge amount of deletions.

4.6.3 Performance of Parallel Algorithms

Table 4.4 shows the performance comparison between our parallel algorithm (Max) and the existing parallel algorithm (PIPA) when applying both of them to our incremental analysis SHARP for different context-sensitive pointer analyses. This table reports the same attributes as Table 4.3. Since PIPA does not have the same parallelization as shown by Section 4.5.1, we use "-" to indicate no such data.

SHARP(Max) indicates 0.6x faster on average over SHARP(PIPA), and 1.3x faster over SHARP. This proves that our parallel algorithm is more faster than the existing parallel algorithm. The actual speedup for each commits are quite different due to the distribution of code changes: more parallelization can be utilized if a commit contains multiple code changes in different methods.

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

4.6.4 Discussions

How does k affect the performance of SHARP. The performance difference of SHARP between 2-CFA and 1-CFA is different from the one between 2-obj and 1-obj. This is affected not only by the value of k, but also by the context-sensitive algorithm (*i.e.*, use a call site or a receiver object). According to the statistics in Table 4.2, 2-CFA always creates much larger sizes of PAG over 1-CFA when comparing the PAG sizes created by 2-obj over 1-obj. For example, 2-CFA computes 1,094,819,728 edges for Lucene, which is about 14x more than the one computed by 1-CFA; while 2-obj only introduces 12,395,079 edges, which is around 1.5x more than the one from 1-obj. Hence, *k-CFA* essentially involves more computation over *k-obj* (when both algorithms use the same k). Our incremental rules are based on *k-CFA* and *k-obj*, which performance follows the same trends as the corresponding context-sensitive algorithms.

When it comes to a larger k (*i.e.*, $k \geq 3$), the scalability of *k-CFA* and *k-obj* becomes even worse due to the significantly increased size of constraints. Theoretically, the worst-case complexity can be doubly exponential [169]. Hence, it is impractical to use $k \geq 3$ in real-world scenarios where the whole program pointer analysis probably cannot terminate. This is also why most context-sensitive pointer analysis works use $k = 1$ and $k = 2$ in their evaluations [46, 47, 79, 86, 128]. Consequently, SHARP is required to handle more constraints introduced by a large k for incremental changes, but should still be much faster than re-running the whole pointer analysis.

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

4.7 Summary

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

5. D4: CONCURRENCY DEBUGGING WITH PARALLEL DIFFERENTIAL ANALYSIS [*]

As the development of hardware and the requirement of performance, parallelism exists in almost every modern programs, *e.g.*, apps, database, machine learning tools, smart contracts, web servers. However, concurrency bugs cannot be effectively avoided and solved when writing parallel, concurrent or event-driven programs, which are more difficult to detect and fix for real-world, large applications. There are many real-world cases when race conditions introduce enormous economic loss or even cost human lives, *e.g.*, the infamous DAO Hack [170] leveraged the re-entrancy vulnerability and stole 12.7M Ether (approximately $150M at the time), the delayed Facebook's IPO [171] by Nasdaq caused $13M loss, and Therac-25 Accident [172] exposed patients to lethal doses of radiation.

Most existing research and commercial techniques [29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39], statically or dynamically, detect concurrency bugs by performing (1) pointer analysis or dataflow analysis or ownership analysis to identify the memory locations accessed by different threads in a static way, or collecting execution traces in a dynamic way; (2) lockset analysis to check mutual exclusion locks held by threads; and (3) techniquesto determine happens-before or concurrent relation between two events (*e.g.*, vector clock [173], epoch [31]). All these analyses become imprecise, unscalable, memory- and time-consuming when applying them on large codebase (*e.g.*, Linux kernel, Google Chrome with millions to billions lines of code).

Another common limitation of the existing works 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 pertains.

To address this problem, one promising direction is to detect concurrency bugs *incrementally*

[*]Reprinted with permission from "D4: Fast Concurrency Debugging with Parallel Differential Analysis" by Bozhen Liu and Jeff Huang. PLDI 2018: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, June 2018, Pages 359–373, Copyright 2018 by Bozhen Liu.

in the programming phase. Upon a change in the source code (*i.e.*, statement addition, deletion or modification), instead of exhaustively re-analyzing the whole program, one can analyze the change only and recompute the impact of the change for bug detection by memorizing the intermediate analysis results. This not only provides early feedback to developers (which reduces the cost of debugging), but also enables efficient bug detection by amortizing the analysis cost.

Besides, data race is exacerbated by interactions between threads and events in real-world applications, *e.g.*, Android apps and distributed systems. One practical solution is to unify threads and events by treating them as entry points of code paths attributed with data pointers, which provides enough precision for data race detection when analyzing shared-memory accesses as well as reduces the computation of conducting expensive whole program analysis.

To solve the problem, a series of research projects were conducted, which focus on developing such novel algorithms for both incremental changes and whole programs to statically detect concurrency bugs (data races and deadlock) in a fast, efficient, scalable and precise way. To be specific, we develop several algorithms and tools to gradually speedup the process of static concurrency bug detection as well as reduce the false positive rate to improve the precision. All the evaluations from the above projects achieve promising performance, scalability and precision for large, real-world software, which answers our research question.

5.1 Introduction

Writing correct parallel programs is notoriously challenging due to the complexity of concurrency. 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 [29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39] 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 pertains.

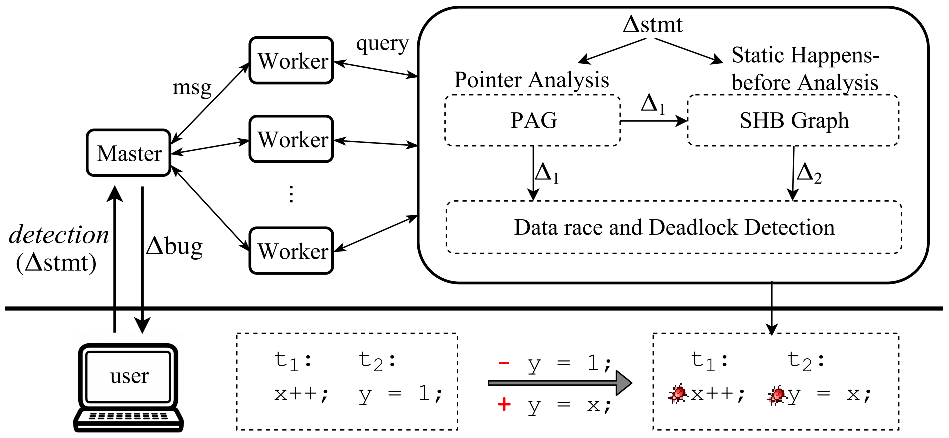


Figure 5.1: Architectural overview of D4. Reprinted with permission from [3].

One promising direction to address this problem is to detect concurrency bugs *incrementally* in the programming phase, as explored by our recent work ECHO [27]. Upon a change in the source code (insertion, deletion or modification), instead of exhaustively re-analyzing the whole program, one can analyze the change only and recompute the impact of the change for bug detection by memoizing the intermediate analysis results. This not only provides early feedback to developers (which reduces the cost of debugging), but also enables efficient bug detection by amortizing the analysis cost.

Despite the huge promise of this direction, a key challenge is how to scale to large real-world applications. Existing incremental techniques are still too slow to be practical. For instance, in our experiments with a collection of large applications from the DaCapo benchmarks [155], ECHO takes over half an hour to analyze a change in many cases. A main drawback is that existing incremental algorithms are either inefficient or inherently sequential. In addition, the existing tool runs entirely in the same process as the integrated development environment (IDE), which severely limits the performance due to limited CPU and memory resources.

In this chapter, we propose D4, a fast concurrency analysis framework that detects concurrency bugs (*e.g.*, data races and deadlocks) *interactively* in the programming phase. D4 advances IDE-based concurrency bug detection to a new level such that it can be deployed non-intrusively in the development environment for read-world large complex applications. D4 is powered by two significant innovations: a novel system design and two novel incremental algorithms for concurrency analysis.

At the system design level, different from existing techniques which integrate completely into the IDE, D4 separates the analysis from the IDE via a client-server architecture, as illustrated in Figure 5.1. The IDE operates as a client which tracks the code changes on-the-fly and sends them to the server. The server, which may run on a high-performance computer or in the cloud with a cluster of machines, maintains a change-aware data structure and detects concurrency bugs incrementally upon receiving the code changes. The server then sends the detection results immediately back to the client, upon which the client warns the developer of the newly introduced bugs or invalidates existing warnings.

At the technical level, D4 is underpinned by two parallel incremental algorithms, which embrace both *change* and *parallelism* for pointer analysis and happens-before analysis, respectively. We show that these two fundamental analyses for concurrent programs, if designed well with respect to code changes, can be largely *parallelized* to run efficiently on parallel machines. As shown in Figure 5.1, D4 maintains two change-aware graphs: a pointer assignment graph (PAG) for pointer analysis and a static happens-before (SHB) graph for happens-before analysis. Upon a set of code changes, Δ_{stmt}, the PAG is first updated and its change Δ_1 is propagated further. Taking Δ_{stmt} and Δ_1 as input, the SHB graph is then updated incrementally and its change Δ_2 together with Δ_1 are propagated to the bug detection algorithms.

D4 can be extended to detect a wide range of concurrency bugs incrementally, since virtually all interesting static program analyses and concurrency analyses rely on pointer analysis and happens-before analysis. For example, the same race checking algorithm in ECHO can be directly implemented based on the SHB graph, and deadlock detection can be implemented by extending D4 with a lock-dependency graph, which simply tracks the lock/unlock nodes in the SHB graph. D4 can also be extended to analyze pull requests in the cloud. For example, in continuous integration of large software, D4 can speed up bug detection by analyzing the committed changes incrementally.

We have implemented both data race detection and deadlock detection in D4 and evaluated its performance extensively on a collection of real-world large applications from DaCapo. The experiments show dramatic efficiency and scalability improvements: by running the incremental analyses on a dual 12-core HPC server, D4 can pinpoint concurrency bugs within 100ms upon a statement change on average, 10X-2000X faster than ECHO and over 2000X faster than exhaustive analysis.

We note that exploiting change and parallelism simultaneously for concurrency analysis incurs significant technical challenges with respect to performance and correctness. Although previous research has exploited parallelism in pointer analyses [87, 88, 90, 93, 94], change and parallelism have never been exploited together. All existing parallel algorithms assume a static whole program and cannot handle dynamic program changes. D4 addresses these challenges by carefully decomposing the entire analysis into parallelizable graph traversal tasks while respecting task dependencies and avoiding task conflicts to ensure the analysis soundness.

In sum, this paper makes the following contributions:

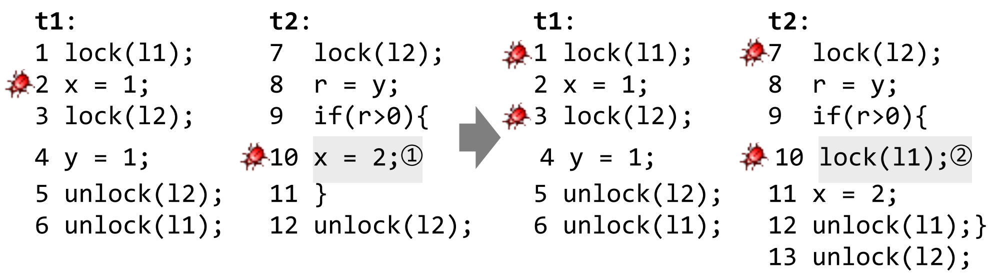
- We present the design and implementation of a fast concurrency analysis framework, D4, that detects data races and deadlocks interactively in the IDE, *i.e.*, in a hundred milliseconds on average after a code change is introduced into the program.

- We present an extensive evaluation of D4 on real-world large applications, demonstrating significant performance improvements over the state-of-the-art.

- D4 is open source. All source code, benchmarks and experimental results are publicly available at https://github.com/parasol-aser/D4.

5.2 Motivation and Challenges

In this section, we first use an example to illustrate the problem and the technical challenges. Then, we introduce existing algorithms and discuss their limitations.



```
t1:              t2:
1 lock(l1);      7  lock(l2);
2 x = 1;         8  r = y;
3 lock(l2);      9  if(r>0){
4 y = 1;         10 x = 2;①
5 unlock(l2);    11 }
6 unlock(l1);    12 unlock(l2);
```

(a)

```
t1:              t2:
1 lock(l1);      7  lock(l2);
2 x = 1;         8  r = y;
3 lock(l2);      9  if(r>0){
4 y = 1;         10 lock(l1);②
5 unlock(l2);    11 x = 2;
6 unlock(l1);    12 unlock(l1);}
                 13 unlock(l2);
```

(b)

Figure 5.2: A motivating example. (a) a data race between lines (2,10) is detected by D4 when Change ① at line 10 is introduced; (b) a new deadlock between lines (1,3;7,10) is detected by D4 when Change ② at lines 10 and 12 is introduced which attempts to fix the race. Reprinted with permission from [3].

5.2.1 Problem Motivation

Consider a developer, Amy, who is working on the Java program shown in Figure 5.2(a). The program consists of two threads `t1` and `t2`, and two shared variables `x` and `y`. As soon as Amy inserts a write ① `x=2` to `t2` and saves the program in the IDE, D4, which runs in the background, will prompt a data race warning on lines 2 and 10, similar to syntax error checking.

As Amy sees the warning, she can analyze and fix the bug immediately without waiting until it is found by a test or a code reviewer, or the bug happens in production. To eliminate the data race, Amy might want to introduce a lock `l1` to protect the write to `x` at line 10. This fixes the data race, however, it introduces a deadlock between the lock pairs at lines (1,3; 7,10) in Figure 5.2(b). Nevertheless, this deadlock is again instantly reported by D4 to guide Amy to fix the bug.

To realize D4 as above, there are three requirements:

1. We need to identify the two threads, the shared data `x` and `y`, and the two locks `l1` and `l2`, *i.e.*, they refer to different locks.

2. We need to identify that the operations by the two threads can execute in parallel, *i.e.*, one does not always happen before the other, and the four lock operations may have circular dependencies.

3. To not interrupt Amy, D4 must be very fast, *i.e.*, it finishes within a sub-second time.

For the first two requirements, we need a pointer analysis and a happens-before analysis. For the third requirement, we must develop an efficient algorithm that can leverage these analyses to detect data races and deadlocks.

5.2.2 Existing Algorithms

Previous work [27] has proposed sequential incremental pointer analysis and happens-before algorithms for data race detection. Although these incremental algorithms are much more efficient than the exhaustive analysis, they are not efficient enough for large software. We already discussed incremental pointer analysis techniques in previous section 2.1.1 and chapter 3, 4, and will focus on existing incremental happens-before analysis in this section.

Table 5.1: Nodes in the SHB graph. Reprinted with permission from [3].

Statements	**Nodes**
❶* $x = y.f$	$write(x), \forall O_c \in pts(y) : read(O_c.f)$
❷* $x.f = y$	$read(y), \forall O_c \in pts(x) : write(O_c.f)$
❸ $synchronized(x)\{\ldots\}$	$\forall O_c \in pts(x) : lock(O_c), unlock(O_c)$
❹ $o.m(\ldots)$	$\forall O_c \in pts(o) : call(O_c.m)$
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 represent array read ($x = y[i]$) and write ($x[i] = y$), resp.

5.2.2.1 Incremental Happens-Before Analysis

The existing technique [27] uses a static happens-before (SHB) graph to compute happens-before relation among abstract threads, memory accesses, and synchronizations. The SHB graph for Java programs is constructed incrementally following the rules in Table 5.1. Among them, statements ❹ (method $call$), ❺ (thread $start$) and ❻ (thread $join$) generate additional edges according to Table 5.2. The SHB graph is represented by sequential traces containing per-thread

Table 5.2: Edges in the SHB graph. Reprinted with permission from [3].

Statements	Edges
❹ $x = o.m(\ldots)$	$\forall O_c \in pts(o) : call(O_c.m) \rightarrow FirstNode(O_c.m)$
	$LastNode(O_c.m) \rightarrow NextNode(call)$*
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): the consecutive node of the method call statement.

nodes in the SHB graph following the program order, connected by inter-thread happens-before edges. For race detection, the happens-before relation between nodes from different threads can then be computed by checking the graph reachability.

Large SHB graph A crucial limitation of this approach is that for large software it can produce a prohibitively large SHB graph. During the graph construction, when a method is invoked, it has to analyze 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. Existing technique 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, all the matching nodes in the graph must be tracked and updated. For large software, this incurs significant repetitive computation because a changed method can be invoked many times.

5.3 Parallel Incremental Happens-Before Analysis

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. Instead of constructing 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. The happens-before relation of nodes (*e.g.*, in the multiply-visited methods) is then computed

Table 5.3: Edges in the new SHB graph. Reprinted with permission from [3].

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

"on-the-fly" following the method-call edges and the inter-thread edges. When a change in a multiply-visited method happens, different node instances corresponding to the change can thus have different happens-before edges without sacrificing accuracy.

5.3.1 SHB Graph Construction

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 that 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 $\langle tar, subshb_{tar}\rangle$ to the $exist$ map and include $ctid$ into the field $tids$ of $subshb_{tar}$. Afterwards, we extract the statements in $target$ and create SHB nodes according to Table 5.1 for each statement and insert it into $subshb_{id}.trace$.

The new happens-before edges are constructed according to Table 5.3. Each edge is labeled with the corresponding 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 avoids 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 existing 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

```
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 5.3: An example for the SHB graph construction. Reprinted with permission from [3].

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.

Example We use an example in Figure 5.3 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 5.4.

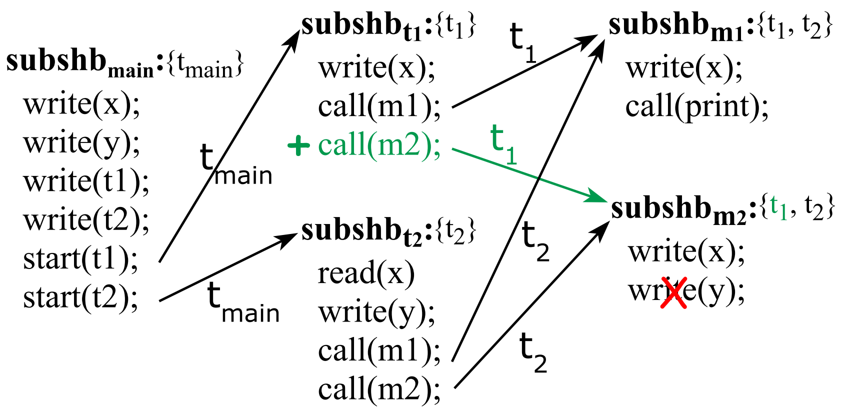


Figure 5.4: The SHB graph for the example in Figure 5.3. Reprinted with permission from [3].

5.3.2 Incremental Graph Update

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; 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 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 $subshb_{tar}$.

Example Consider two changes in our example in Figure 5.3: (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 $m2()$. We add an edge $call(m2) \xrightarrow{t_1} subshb_{m_2}$ to the graph and add t_1 into $subshb_{m_2}.tids$. For (ii), we localize the $write(y)$ node corresponding to this statement and simply remove it from $subshb_{m_1}$.

5.3.3 Computing Happens-Before Relation

Our new SHB graph representation also makes computing the HB relation more efficient than existing approach [27]. For changes in a method invoked multiple times, instead of checking the path reachability between each individual pair of nodes, we can check for multiple node pairs altogether. For example, in Figure 5.4 although the method $m2()$ is invoked once by t_1 and once by 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 the 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.

5.4 Distributed System Design

There are three main components in our design of distributing the analysis to a remote server, which is expected to have more computing 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 also sends the detection results to the IDE. We next focus on describing the second component, which is the core of our system.

Parallel Analysis Framework We implement a communication interface between the client and the server based on the open-source Akka framework [174], 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 all tasks are processed.

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 [175], 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 from each id to subgraph.

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 pointer assignment edge.

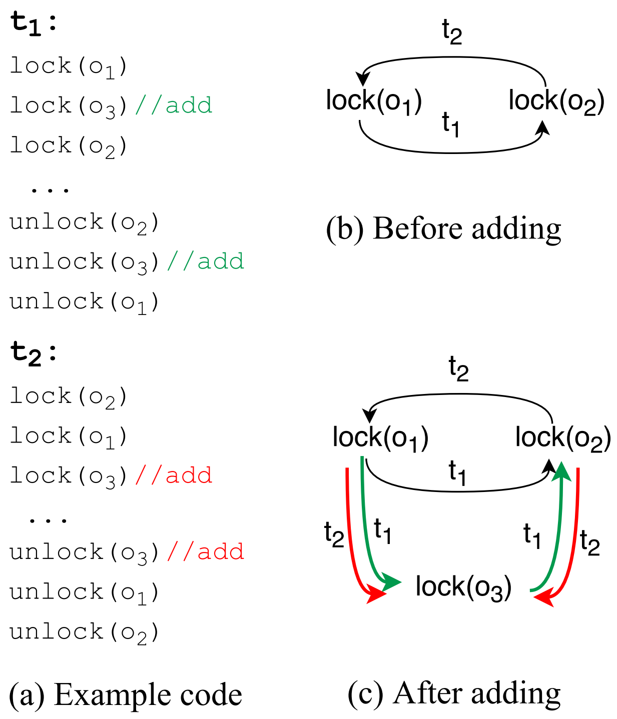
5.4.1 Connection with Dynamic Graph Algorithms

D4 updates the two graphs (PAG and SHB) dynamically, which is related to dynamic algorithms on directed graphs. Existing dynamic graph algorithms have focused on shortest path [176], transitive closure [157, 176] and max/min flow [177]. For pointer analysis, our priority here is to efficiently update the points-to sets of a specific set of nodes in the PAG. For happens-before analysis, the problem is to effectively update the content of each node ($subshb$) as well as its affected nodes/edges based on the definition of the happens-before relation. Although existing algorithms cannot be directly applied to our cases, for certain tasks (*e.g.*, SCC detection and checking reachability from a pointer node to an object node) we may utilize dynamic reachability algorithms [157, 176] to improve the performance.

5.5 D4: Concurrency Bug Detection

D4 can be used to develop many interesting incremental concurrency analyses, such as detecting data races, atomicity violations and deadlocks.

We have implemented both data race and deadlock detection in D4. Our race detection checks the happens-before relation and the lockset condition between every conflicting pair of *read* and *write* nodes on the same abstract heap from different threads. If the two nodes cannot reach each other in the SHB graph and there is no common lock protection, we will report them as a race. Our race detection algorithm is the same as that presented in [27], except that we use a different SHB



```
t1:
lock(o1)
lock(o3)//add
lock(o2)
  ...
unlock(o2)
unlock(o3)//add
unlock(o1)
t2:
lock(o2)
lock(o1)
lock(o3)//add
  ...
unlock(o3)//add
unlock(o1)
unlock(o2)
```

(a) Example code

(b) Before adding

(c) After adding

Figure 5.5: An example for the LD graph construction. Reprinted with permission from [3].

graph representation to determine the happens-before relation.

In this section, we focus on our novel incremental deadlock detection algorithm. 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. Besides, the ability to detect deadlocks is particularly important for interactive race detection tools, because once a data race is detected, programmers often use locks to fix the race, which may introduce new deadlock bugs.

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.

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_2, 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 5.5 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.

Incremental Deadlock Detection Our basic idea of incremental deadlock detection is to look for cycles in the LD graph with edge labels from multiple threads, which indicate circular dependencies of locks. We then check the happens-before relation between the involved nodes to find real deadlocks. For example, in Figure 5.5(b), $lock(o_1) \xrightarrow{t_1} lock(o_2)$ and $lock(o_2) \xrightarrow{t_2} lock(o_1)$ form a circular dependency. To realize incremental deadlock detection, we develop an incremental algorithm for updating the LD graph and an incremental algorithm for deadlock checking.

Incremental LD Graph Update For an added synchronized statement in thread t, we first locate the method it belongs to 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 of LD graph 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 traverse the LD graph in the reverse order to discover the incoming $lock$ nodes of $pred$ with edges labeled t. For each such node $pred'$, 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 5.5(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: $lock(o_1) \xrightarrow{\text{t1}}$

```
Algorithm 8: IncrementalDeadlockDetection
    Global States: $shb$ - updated SHB graph
                   $ldg$ - updated LD graph
    Input        : $\Delta_{lock}$ - the changed lock nodes
    Output       : $deadlocks$ - detected deadlocks
 1  $cycles \leftarrow$ DiscoverCircularDependency($ldg$, $\Delta_{lock}$)
 2  foreach $c \in cycles$ do
 3      ParallelDeadlockDetection($c$)
 4  end
 5  ParallelDeadlockDetection($c$):
 6  $tids \leftarrow$ ExtractTidsInCycle($c$)
 7  foreach ($t_i$, $t_j$) $\in tids$ do
        // for each pair of threads
 8      $lock(x), lock(y) \leftarrow$ 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          $deadlocks \leftarrow c$
11      end
12  end
```

$lock(o_3)$, $lock(o_1) \xrightarrow{t2} lock(o_3)$, $lock(o_3) \xrightarrow{t1} lock(o_2)$ and $lock(o_2) \xrightarrow{t2} lock(o_3)$, as shown in Figure 5.5(c).

Incremental Deadlock Checking Algorithm 8 illustrates the incremental deadlock detection. The key idea is to check only the cycles containing the changed (added or deleted) $lock$ nodes. We first collect all the circular dependencies that include the changed $lock$ nodes. Then, we parallelize deadlock detection for all cycles by checking the happens-before relation between conflicting lock nodes from different threads in each cycle.

5.6 Evaluation

We implemented D4 based on the WALA framework [154] and evaluated it on a collection of 14 real-world large Java applications from DaCapo-9.12, as shown in Table 5.4. 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) proces-

Table 5.4: Benchmarks and the PAG metrics. Reprinted with permission from [3].

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
xalan	21K	33K	576K	15K	138M

sors. In this section, we report the results of our experiments.

Evaluation Methodology For each benchmark, we run three sets of experiments. (1) We first run the whole program exhaustive 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 configurations: on the local client machine with a single thread (*D4-1*) to evaluate our incremental algorithms only, and on the server machine with 48 threads (*D4-48*) to evaluate our parallel incremental algorithms. We measure the time taken by each component in each step and compare the performance between the exhaustive analysis and D4. In addition, we repeat the same experiments for ECHO running on the client machine to compare the performance between D4 and ECHO.

Table 5.5: Performance of concurrency bug detection. Reprinted with permission from [3].

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	16ms	2min	231ms	2min	23ms	32s
batik	>6h	5.2min	2h	1.3s	13min	0.9s	57s	1ms	11ms	1ms	8ms
eclipse	1.2h	5s	10min	0.3s	5min	152ms	49s	110ms	2min	13ms	4s
h2	4h	1.2s	6min	33ms	39s	12ms	15s	1ms	18ms	1ms	10ms
jython	3.3h	1s	5min	19ms	20s	17ms	11s	0.4ms	242ms	1ms	53ms
luindex	3h	43ms	2min	4ms	7s	1.9ms	3.8ms	32ms	29s	25ms	17s
lusearch	2.6h	19ms	1.7min	7ms	5s	2.2ms	4.1ms	1ms	3ms	1ms	1.3s
pmd	0.8h	3.1s	7min	41ms	9s	6.8ms	1s	5.4ms	1s	0.2ms	53ms
sunflow	3.6h	1s	3min	0.15ms	23ms	0.1ms	12ms	0.3ms	8ms	0.1ms	2ms
tomcat	0.7h	1.7s	6min	6ms	4.3s	1.5ms	0.82s	0.1ms	0.9ms	0.1ms	0.4ms
tradebeans	0.8h	49ms	3min	1.1ms	1s	0.8ms	0.3s	0.1ms	1.3ms	0.1ms	0.4ms
tradesoap	0.9h	47ms	2.6min	0.9ms	1s	0.7ms	0.4s	0.1ms	1ms	0.1ms	0.3ms
xalan	0.5h	33ms	1.8min	0.2ms	42ms	0.1ms	15ms	1ms	2.7ms	0.1ms	1.1ms
Average	>2.6h	25s	21min	1.8s	2.9m	0.12s	20s	29ms	21s	5ms	4.2s

Benchmarks The metrics of the benchmarks and their PAGs are reported in Table 5.4. Columns 2-6 report the numbers of classes, methods, pointer nodes, object nodes and edges in the PAGs, respectively. More than half of the benchmarks contain over 1M pointer nodes and over 200M edges 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. This exclusion makes a trade-off between soundness and computational cost, which is a common practice for both static and dynamic analysis tools to improve performance.

5.6.1 Performance of Concurrency Bug Detection

Table 5.5 reports the performance of concurrency bug detection for all the 13 multithreaded applications in DaCapo-9.12 (`fop` is excluded because it is single-threaded), 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 whole program. The time for ECHO and D4 includes that taken by incremental algorithms for updating the graphs (*i.e.*, SHB and LD) and detecting bugs per change.

Overall, the exhaustive analysis requires a long time (>2.6h 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 10X-2000X speedup for all cases on average, and 5X-50X speedup for the worst cases. ECHO takes 25s on average and 21min in the worst case to detect data races upon a change, while *D4-1* and *D4-48* take only 1.8s and 0.12s respectively on average, and 2.9min and 20s in the worst case. The incremental deadlock detection in D4 is also very efficient. It takes less than 29ms on average and 21s in the worst case for *D4-1*, and 5ms and 4.2s for *D4-48* per change. Compared to the exhaustive analysis, it is over 2000X faster.

Performance weakness of the new SHB analysis For small programs (*e.g.*, <50 LOC), the new SHB analysis may require more time than the previous SHB analysis [27] to compute for incremental updates. There are two main reasons: (1) there are fewer repetitive method calls in small programs, hence the new SHB representation cannot be fully utilized; (2) the construction of the new SHB graph is more complex (*e.g.*, maintenance of maps and subgraph fields), which leads to a trade-off between program size and performance.

5.6.2 Discussions

Network traffic time We also measured the network traffic time of the server mode in D4. In our lab environment with a standard wireless connection, the network traffic time is under 0.1ms per statement change, hence it is negligible.

Scalability We notice that the scalability of parallel incremental pointer analysis cannot catch up with that of parallel concurrency bug detection, due to two main reasons: (1) we only process one edge in WL (lines 3-5 of Algorithm 2) per iteration in order to avoid conflict of edge updates; (2)

the shape of the PAG determines the utilization of the parallel resources. For a deletion, if the chain of dependent variables of the change node is long and the in-degree of the variables on the chain is large, but the out-degree is small, our algorithm cannot scale well on this pattern, because most of the work has to be done sequentially, such as checking a large number of incoming neighbours and updating the affected edges.

Bug detection precision We note that 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.

We also studied the detection results reported by the whole program race detector in ECHO and D4, and confirmed that they report the same results. However, without significant knowledge in the application code it is difficult to verify the reported warnings (if they are true bugs or false alarms). 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.

D4 batch mode Although in our experiments D4 is evaluated for each single statement change (to avoid any biases caused by choosing a random set of changes), it is unnecessary to run D4 after every line of change, but D4 can be executed after a batch of changes. Currently, D4 runs whenever a file is saved by the user in the IDE, or the user can trigger D4 whenever an incremental check is necessary. The size of a batch varies in different applications but is typically small. For example, for good quality real-world projects such as `h2` and `eclipse`, we observe that most of the commits contain only 1-50 lines of code changes. Besides, D4 can also run entirely on a single local machine to eliminate the cost of message passing over network.

Complex code changes As an IDE-based tool, we focus on source-level (i.e., Java bytecode) analysis. It is difficult for static analysis to handle link-time changes (such as dynamic libraries), because they only get into effect at integration time. We leave link-time changes for future research. Also, currently we do not handle package-level changes such as `import`. If a package is swapped out we simply re-build the PAG. We note that the analysis is only triggered after the program type checks. For changes that result in type errors, *e.g.*, missing a class or method definition, they are handled by the type checker in the IDE. D4 is based on Andersen's algorithm, which does not deal with class-escape information. Hence, we analyze constraints from program changes without considering the modifiers.

Practicability Although we did not evaluate D4 in a production environment where even larger programs are running without an IDE, the fundamental and scalable techniques we provide can be utilized by other analysis tools since we aim at source code analysis. Besides, it is possible to make D4 independent of the IDE based on the Language Server Protocol [178], which we leave for future research.

5.7 Summary

We have presented a novel framework for detecting concurrency bugs efficiently in the programming phase. Powered by a distributed system design and new parallel incremental algorithms, D4 achieves dramatic performance improvements over the state-of-the-art. Our extensive evaluation on real-world large systems demonstrates excellent scalability and efficiency of D4, which is promising for practical use.

6. O2: EFFICIENT AND PRECISE STATIC RACE DETECTION WITH ORIGINS *

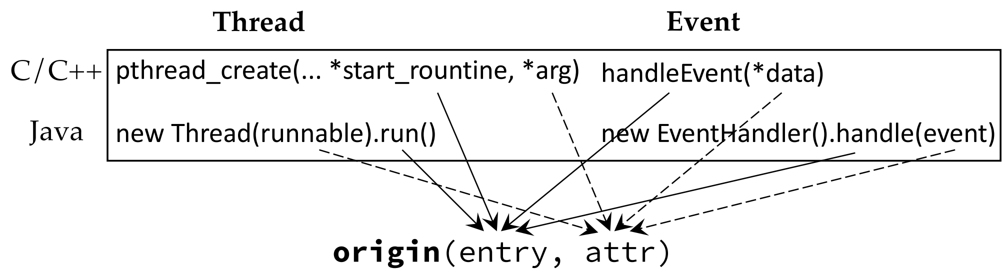
Data races are among the worst bugs in software in that they exhibit non-deterministic symptoms and are notoriously difficult to detect. The problem is exacerbated by interactions between threads and events in real-world applications. We present a novel static analysis technique, O2, to detect data races in large complex multithreaded and event-driven software. O2 is powered by "origins", an abstraction that unifies threads and events by treating them as entry points of code paths attributed with data pointers. Origins in most cases are inferred automatically, but can also be specified by developers. More importantly, origins provide an efficient way to precisely reason about shared memory and pointer aliases.

Together with several important design choices for race detection, we have implemented O2 for both C/C++ and Java/Android applications and applied it to a wide range of open-source software. O2 has found new races in every single real-world code base we evaluated with, including Linux kernel, Redis, OVS, Memcached, Hadoop, Tomcat, ZooKeeper and Firefox Android. Moreover, O2 scales to millions of lines of code in a few minutes, on average 70x faster (up to 568x) compared to an existing static analysis tool from our prior work, and reduces false positives by 77%. We also compared O2 with the state-of-the-art static race detection tool, RacerD, showing highly promising results. At the time of writing, O2 has revealed more than 40 unique previously unknown races that have been confirmed or fixed by developers.

6.1 Introduction

Threads and events are two predominant programming abstractions for modern software such as operating systems, databases, mobile apps, and so on. While the thread vs. event debate has never ended [104, 106], it is clear that both face a common problem: threads and events often lead to non-deterministic behaviors due to various types of race conditions, which are notoriously

*Reprinted with permission from "When Threads Meet Events: Efficient and Precise Static Race Detection with Origins" by Bozhen Liu and Peiming Liu and Yanze Li and Chia-Che Tsai and Dilma Da Silva and Jeff Huang. PLDI 2021: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, June 2021, Pages 725–739, Copyright 2021 by Bozhen Liu.



Thread
Event
C/C++
Java
pthread_create(... *start_rountine, *arg)
handleEvent(*data)
new Thread(runnable).run()
new EventHandler().handle(event)
origin(entry, attr)

Figure 6.1: An "origin" view of threads and events. Reprinted with permission from [4].

difficult to find, reproduce, and debug.

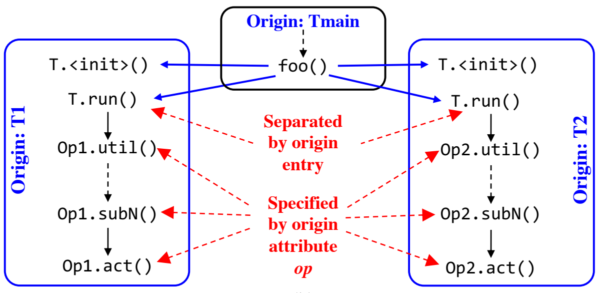
There has been intensive research on race detection of multithreaded code. Most successful techniques have been dominated by dynamic analysis [31, 33, 179, 180, 181, 182], notably Google's ThreadSanitizer [37]. However, dynamic techniques face an inherent challenge of performance overhead and low code coverage. In contrast, static detection techniques have had only very limited success, notably Facebook's RacerD [133, 138], despite decades of research [30, 34, 38, 183, 184]. A crucial reason is that reasoning about races typically requires sophisticated pointer alias analysis to attain accuracy, which is difficult to scale.

Races in event-driven programs have attracted much attention in recent years [139, 140, 141, 142, 143, 144, 145, 146]. Event-based races can be more challenging to detect than thread-based races because most events are asynchronous and the event handlers may be triggered in many different ways. Moreover, the difficulty in detecting event-based races is exacerbated by interactions between threads and events, which are common in real-world software such as distributed systems. The state-of-the-art race detectors [37, 133, 138] do not perform well in detecting event-based races, also due to the large space of casual orders among event handlers and threads.

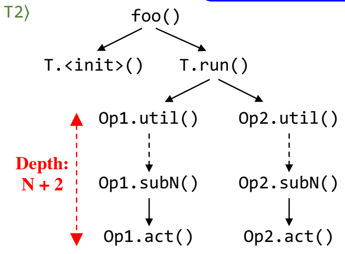
In this paper, we present O2, a new system for detecting data races in complex multithreaded and event-driven applications. We show that conventional thread-sensitive static analysis (with some tuning and care) is highly effective for finding races, even more effective than RacerD. A key concept behind O2 are *origins*, an extended notion of threads and events that unify them through two parts: 1) *an entry point* that represents the beginning of a thread or an event handler, and 2)

```
public void foo(){//main thread
  Obj s = new Obj();//o1
  Op op1 = new Op1();//o2
  Op op2 = new Op2();//o3
  new T(s, op1).run();//o4 → Origin: T1
  new T(s, op2).run();//o5 → Origin: T2
}
public class T extends Thread{
  Obj f; Op op;//super class of Op1 and Op2
  public T(Obj a, Op b){
    f = a; op = b; }
  public void run(){
    op.util(f, new Obj());//o6
  } //with origin:⟨o6,T1⟩ and ⟨o6,T2⟩
 }
void util(Obj x, Obj y){
  sub1(x, y); ...
}
void sub1(Obj x, Obj y){
  sub2(x, y); ...
}
                    ...
void subN(Obj x, Obj y){
   y.do_something();
   act(x, y);}
```

(a)



(b)



(c)

Objects	Accessed by	Meaning
⟨o6,T1⟩	T1	Local to Origin T1
⟨o6,T2⟩	T2	Local to Origin T2
⟨o1,Tmain⟩	T1&T2	Allocated by Origin Tmain, Shared by Origins T1 and T2

(d)

Figure 6.2: (a) The example code. (b) The origin-sensitive call graph, where each origin consists of a sequence of calls of arbitrary length. The origin attributes precisely determine the call chain executed in each origin. (c) The context-sensitive call graph without origin. (d) A sample origin-sharing analysis (OSA) output. Reprinted with permission from [4].

a set of attributes that capture additional semantics, such as thread ID, event type, or pointers to memory objects that will be used in the thread or event handler. Figure 6.1 depicts an "origin" view for threads and events in C/C++ and Java. The origin attributes can be specified or inferred automatically at the origin's entry point and the allocation site of the receiver object. We elaborate the design in Section 6.2.1.

Rather than a straightforward unification, origins enables *origin-sensitive pointer analysis* (**OPA**), in which the conventional call-string-based or object-based context abstractions are replaced by origins. This has several advantages:

- Functions within the same origin share the same context, therefore the computation complexity inside an origin does not grow with the length of the call chain; and

- Computing k-most-recent calling contexts at every call site is redundant in many applications [129], *e.g.*, when determining which objects are local to or are shared by which threads.

- The crucial origin entry point is preserved, not discarded as a trivial context in *k-limiting* [127] when the call stack's depth exceeds the context depth k.

Meanwhile, compared to conventional thread-based [117, 118, 119, 120] or event-based [111, 112, 113, 114, 115, 116] analyses, the inclusion of data pointers in origins enables precisely identifying shared- and local-memory accesses by different threads and events. We develop *origin-sharing analysis* (**OSA**), which uses an origin-sensitive heap abstraction to precisely compute heap objects local to each origin, and objects shared by each combination of multiple origins. OSA has several advantages over classical thread-escape analysis. In particular, besides answering *whether* an object is shared, OSA provides detailed information on *how* the object is shared across origins, which is needed by race detection.

To illustrate these advantages, consider an example in Figure 6.2. To correctly infer that threads `T1` (line 5) and `T2` (line 6) do not access the same data on line 23, typically, a k-call-site analysis (denoted *k-CFA*) is performed, in which k is the depth of the call chain [126]. Additionally, a call-site-sensitive heap context is necessary to analyze the object allocation on line 13. This complexity is shared by k-object-sensitivity [49] (denoted *k-obj*), in which the sequence of `subN()` functions are invoked on different receiver objects. With origins, it suffices to mark the function `run()` in each thread as an origin's entry point. In this way, the allocation on line 13 can be distinguished by an origin unique to its thread. At the same time, the virtual function `act()` invoked by each thread on line 24 can be distinguished by the origin's data pointers: *s* and *op1* for `T1`, and *s* and *op2* for `T2`. Thus, it can be inferred that the two threads invoke different member functions (from `util()` to `act()`) in classes `Op1` and `Op2` respectively, which manage the object that *y* points to differently. Figure 6.2(d) shows a sample OSA output for the example code.

In addition to OPA and OSA, there are a few important design choices we made in O2 that together make static race detection highly effective. First, O2's race detection engine is highly optimized to achieve scalability and precision. We construct a static happens-before graph (SHB) and use static "happens-before" instead of static "may-happen-in-parallel" as the foundational concept of the analysis. This allows pruning many infeasible race pairs by checking only graph reachability. Second, we develop several sound optimizations that scale race detection to large code bases, including:

- An efficient representation of origin-local happens-before relations, which further enables efficient checking and caching the happens-before relation between memory accesses;
- A compact representation of locksets, which enables a fast check of common locks and an efficient cache policy of the intermediate results;
- A lock-region-based race detection that allows effectively merging many memory accesses into a representative one, which reduces the number of race checks significantly.

We implemented O2 for both C/C++ and JVM applications based on LLVM [185] and WALA [154], and applied it to a large collection of widely-used mature open-source software. The results show that O2 is both efficient and precise: it scales to large programs, being able to analyze millions of lines of code in a few minutes, up to 568x faster and reduces false positives by 77% on average compared to existing static analyses from our prior work (D4 [3]).

We compared O2 with RacerD (v1.0.0), the most recent state-of-the-art static data race detector. For the programs that can be compiled and analyzed by RacerD, O2 achieves comparable performance while detecting many new races and 4.33x fewer warnings on average. In most of the evaluated programs in which O2 detects new races, RacerD either fails to find the races or cannot run due to compiler errors.

Surprisingly, O2 found real and previously unknown races in every single real-world code base we evaluated with. At the time of writing, O2 has revealed more than 40 unique race bugs that have been confirmed or fixed by developers, including Linux kernel, Redis/RedisGraph, Open vSwitch OVS, Memcached, Hadoop, ZooKeeper, and the Firefox Android apps. O2 has been integrated into a commercial static analyzer.

6.2 Origin-Sensitive Analyses

In this section, we first present origins, OPA and OSA. The use of OPA and OSA enables a more precise pointer analysis and identification of shared- and local-memory accesses by threads

Table 6.1: The origin entry points identified by O2. Reprinted with permission from [4].

Threads	Event handlers
java.lang.Thread.start()	actionPerformed(. . .)
java.lang.Runnable.run()	onMessageEvent(. . .)
java.util.concurrent.Callable.call()	handleEvent(. . .)
pthread_create(. . .)	onReceive(. . .)

* More details for Android events are in Section 6.3.2.

and events. Beyond race detection, OPA and OSA can benefit any analysis that requires analyzing pointers or ownership of memory accesses, *e.g.*, deadlock, over-synchronization, and memory isolation. We present O2's race detection engine in the next section.

6.2.1 Automatically Identifying Origins

In general, a program can be divided into many different origins, each represents a unit of the program's functionality. At the code level, an origin is a set of code paths all with the same starting point (*i.e.*, the entry point) and data pointers (*i.e.*, the origin attributes). In this way, origins divide a program into different sets of code paths according to their semantics where each origin represents a separate semantic domain. While origins can be specified by code annotations, we aim to extract them automatically from common code patterns in multithreaded and event-driven programs. Our system identifies two kinds of origins automatically by default: threads and event handlers. Finding static threads is not difficult in practice because threads are almost always explicitly defined, either at the language level or through common APIs such as POSIX Threads (Pthreads) and `Runnable` and `Callable` interfaces in Java. Finding event handlers relies on code patterns such as Linux system call interfaces (all with prefix `__x86_sys_`), Android callbacks (`onReceive` and `onEvent`), and popular even-driven frameworks (Node.js and REST APIs). In cases where threads or events are implicit, such as customized user-level threads, developers may be willing to provide annotations to mark origins, since customized threads are likely to be an important aspect of the target application.

For Java and Pthread-based C/C++ programs, we automatically identify the methods in Ta-

Table 6.2: The OPA rules for Java. Consider the following statements are in method `m()` with Origin $\mathbb{O}_i$, denoted $\langle m, \mathbb{O}_i \rangle$. The edges $\rightarrow$ are in the PAG and $\rightarrowtail$ in the call graph. Reprinted with permission from [4].

Statement	**Pointer Assignment Edge & Call Edge**
❶ $x = new\ C()$	$\langle o, \mathbb{O}_i \rangle \rightarrow \langle x, \mathbb{O}_i \rangle$
❷ $x = y$	$\langle y, \mathbb{O}_i \rangle \rightarrow \langle x, \mathbb{O}_i \rangle$
❸ $x.f = y$	$\dfrac{\forall \langle o, \mathbb{O}_k \rangle \in pts(\langle x, \mathbb{O}_i \rangle)}{\langle y, \mathbb{O}_i \rangle \rightarrow \langle o, \mathbb{O}_k \rangle.f}$
❹ $x = y.f$	$\dfrac{\forall \langle o, \mathbb{O}_k \rangle \in pts(\langle y, \mathbb{O}_i \rangle)}{\langle o, \mathbb{O}_k \rangle.f \rightarrow \langle x, \mathbb{O}_i \rangle}$
❺ $x[idx] = y$	$\dfrac{\forall \langle o, \mathbb{O}_k \rangle \in pts(\langle x, \mathbb{O}_i \rangle)}{\langle y, \mathbb{O}_i \rangle \rightarrow \langle o, \mathbb{O}_k \rangle.*}$
❻ $x = y[idx]$	$\dfrac{\forall \langle o, \mathbb{O}_k \rangle \in pts(\langle y, \mathbb{O}_i \rangle)}{\langle o, \mathbb{O}_k \rangle.* \rightarrow \langle x, \mathbb{O}_i \rangle}$
❼ $x = y.f(a_1, ..., a_n)$ //non-origin entry	$\dfrac{\forall \langle o, \mathbb{O}_k \rangle \in pts(\langle y, \mathbb{O}_i \rangle) \quad \langle f', \mathbb{O}_i \rangle = dispatch(\langle o, \mathbb{O}_k \rangle, f)}{\langle o, \mathbb{O}_k \rangle \rightarrow \langle f'_{this}, \mathbb{O}_i \rangle}$ $\langle a_h, \mathbb{O}_i \rangle \rightarrow \langle p_h, \mathbb{O}_i \rangle$, where $1 \leq h \leq n$ $\langle f'_{ret}, \mathbb{O}_i \rangle \rightarrow \langle x, \mathbb{O}_i \rangle$ ***add call edge*** $\langle m, \mathbb{O}_i \rangle \rightarrowtail \langle f', \mathbb{O}_i \rangle$
❽ $x = new\ O(b_1, ..., b_n)$ //origin allocation	***Compute new origin***: $\mathbb{O}_j$ $\dfrac{\langle init, \mathbb{O}_j \rangle = dispatch(-, init)}{\langle o, \mathbb{O}_j \rangle \rightarrow \langle init_{this}, \mathbb{O}_j \rangle}$ $\langle o, \mathbb{O}_j \rangle \rightarrow \langle x, \mathbb{O}_i \rangle$ $\langle b_h, \mathbb{O}_i \rangle \rightarrow \langle p_h, \mathbb{O}_j \rangle$, where $1 \leq h \leq n$ ***add call edge*** $\langle m, \mathbb{O}_i \rangle \rightarrowtail \langle init, \mathbb{O}_j \rangle$
❾ $x.entry(c_1, ..., c_n)$ //origin entry point	$\dfrac{\forall \langle o, \mathbb{O}_j \rangle \in pts(\langle x, \mathbb{O}_i \rangle) \quad \langle entry', \mathbb{O}_j \rangle = dispatch(\langle o, \mathbb{O}_j \rangle, entry)}{\langle o, \mathbb{O}_j \rangle \rightarrow \langle entry'_{this}, \mathbb{O}_j \rangle}$ $\langle c_h, \mathbb{O}_i \rangle \rightarrow \langle p_h, \mathbb{O}_j \rangle$, where $1 \leq h \leq n$ ***add call edge*** $\langle m, \mathbb{O}_i \rangle \rightarrowtail \langle entry', \mathbb{O}_j \rangle$

ble 6.1 as the origin *entry* points, which are frequently used to run code in parallel or handle an event. We then reason about the origin *attributes* in order to distinguish different origins with the same entry point but different data. The origin attributes can be inferred at two places:

- *Origin Allocation* is the allocation site of a receiver object of an origin entry point. The attributes include the arguments passed to the allocation site. For example, `o4` (line 5) is an origin allocation in Figure 6.2, which is the receiver object of the entry point `start()` of Origin `T1`. As its arguments, *s* and *op1* are the origin attributes of `T1`.

- *Origin Entry Point* may be invoked with parameters, of which pointers are also included in the attributes. For example, `onReceive(context, intent)` is an entry point of `BroadcastReceiver` in Android apps, where *intent* contains the incoming message and *context* represents the environment the message is sent from.

6.2.2 Origin-Sensitive Pointer Analysis

Interestingly, reasoning about pointers and heap objects can be done simultaneously with origin-sensitive pointer analysis (OPA). Pointer analysis typically uses the *pointer assignment graph* (PAG) [42] to represent points-to relations between pointers and objects. To achieve good precision, the PAG constructed by OPA is built together with the call graph (a.k.a. on-the-fly pointer analysis [42]). The key difference is that the context of pointers in OPA is represented by origins. The rules of OPA for Java are summarized in Table 6.2. A set of similar rules can be inferred for other programming languages.

Intra-Origin Constraints Statements ❶-❼ are in met-hod $\langle m, \mathbb{O}_i \rangle$, and all the program elements created by them share the same origin $\mathbb{O}_i$ to indicate where they are originated from. For example, the allocated object by statement ❶ is represented as $\langle o, \mathbb{O}_i \rangle$ and assigned to pointer $\langle x, \mathbb{O}_i \rangle$, and their relation is represented by a points-to edge $\langle o, \mathbb{O}_i \rangle \rightarrow \langle x, \mathbb{O}_i \rangle$ in the PAG .

An object field pointer is distinguished by the origin of its receiver object. For statement ❹, each receiver object $\langle o, \mathbb{O}_k \rangle$ corresponds to an object field pointer $\langle o, \mathbb{O}_k \rangle .f$ that points to $\langle x, \mathbb{O}_i \rangle$. Note that a pointer and its points-to objects may have different origins, which shows how data flows across origins.

Although there exists a large body of work that can infer the content of arrays, analyzing array index *idx* in statements ❺❻ is statically undecidable and expensive. Hence, we do not distinguish

```
1  public static void main(){//Tmain
2    TA a = new TA();//oa→Ta
3    TB b = new TB();//ob→Tb
4    a.start();
5    b.start();
6  }
7  Class TA extends T {
8    TA() { super(); ... }
9  }
10 Class TB extends T {
11   TB() { super(); ... }
12 }
```

```
12 Class T {
13   Object f;
14   T() {
15     f = new Object();
       //without switch: ⟨of,Tmain⟩
       //with context switch: ⟨of,Ta⟩ and ⟨of,Tb⟩
16   }
17   public void run(){
18     f.do_something();
19   }
20 }
```

Figure 6.3: An example to explain why it is necessary to switch context at origin allocations. Reprinted with permission from [4].

different array indexes: array objects are modeled as having a single field $*$ that may point to any value stored in the array, *e.g.*, $x[idx] = y$ is modeled as $x.* = y$. This model simply captures objects allocated by different origins that flow to an array without any complex index analysis. Besides, our algorithm can be easily integrated with existing array index analysis algorithms with no conflict.

A non-origin entry method call ❼ invokes a target method `f'` within the same origin $\mathbb{O}_i$ as its caller, even though its receiver object $\langle o, \mathbb{O}_k \rangle$ might be allocated from a different origin $\mathbb{O}_k$. To determine a virtual call target and its context (*e.g.*, the call on line 13 in Figure 6.2), we use the type of its receiver object o and the origin $\mathbb{O}_i$ of which thread/event-handler executes the target. The target's origin must be consistent with its caller's, regardless of whether it is an entry point or not.

Inter-Origin Constraints We switch contexts from current origin $\mathbb{O}_i$ to a new origin $\mathbb{O}_j$ for an origin allocation ❽ and an origin entry point ❾.

Note that, to avoid false aliasing introduced by thread creations, we analyze every origin allocation in its new origin instead of its parent origin where it should be executed. Figure 6.3 shows two origins (`Ta` and `Tb`) allocated in Origin `Tmain`. The two origin allocations share the same super constructor `T()`. If we analyze them in their parent origin `Tmain`, only one object o_f will be allocated for field f on line 15. This will cause $pts(o_a.f) = pts(o_b.f) = \{\langle o_f, Tmain \rangle\}$, which introduces false aliasing. To eliminate such imprecision, OPA creates two objects, $\langle o_f, Ta\rangle$ and $\langle o_f, Tb\rangle$, for each f under each origin by forcing the context switch at origin allocations on lines 2 and 3.

To identify origin allocations on-the-fly, we check the type of the allocated object against the classes in Table 6.1, *i.e.*, if it implements interface `Runnable` or event handler `handleEvent()`. Context switch on ❽ can efficiently separate data flows to the same origin constructor but from different allocation sites, *e.g.*, both *op1* and *op2* flow to the constructor of `T` in Figure 6.2. Specifically, in this example a new and unique origin $\mathbb{O}_j$ is created for this new allocation $\langle o, \mathbb{O}_j\rangle$.

Both ❽ and ❾ designate the attributes for the new origin $\mathbb{O}_j$, including constructor arguments ($b_1, ..., b_n$) and method parameters ($c_1, ..., c_n$), which reveal significant information of the accessed data and the origin behavior. To reflect the ownership, the actual parameters use $\mathbb{O}_i$ as their contexts and the formal ones use $\mathbb{O}_j$. Meanwhile, call edges are added in the call graph, *e.g.*, $\langle m, \mathbb{O}_i\rangle \rightarrowtail \langle init, \mathbb{O}_j\rangle$ for ❽ and $\langle m, \mathbb{O}_i\rangle \rightarrowtail \langle entry', \mathbb{O}_j\rangle$ for ❾.

Wrapper Functions and Loops In practice, both ❽ and ❾ may be hidden in a wrapper function (*e.g.*, cross-platform thread wrappers) invoked by multiple call sites. To efficiently separate such origins, we can extend the entry point of an origin to also include its *k-call-site*. In our tools, we set k=1. Meanwhile, for an origin allocated in a loop, we always create two origins with identical attributes but different origin IDs.

***K*-Origin-Sensitivity** In the same spirit as k-CFA and k-obj, a sequence of origins can be concatenated, denoted as *k-origin*. For example, a method `m()` can be denoted as follows:

$$\langle m, [\mathbb{O}_1, \mathbb{O}_2, ..., \mathbb{O}_{k-1}, \mathbb{O}_k]\rangle$$

where `m()` is invoked within Origin $\mathbb{O}_k$ that has a parent origin $\mathbb{O}_{k-1}$, etc. k-origin can further improve the precision when a pointer propagates across nested origins, and we observed such cases in many of our evaluated programs (*e.g.*, Redis) where thread creations are nested.

Time Complexity Table 6.3 summarizes the worst-case time complexity of different pointer analysis algorithms according to [186], where *p* and *h* are the number of statements and heap

Algorithm 9: Origin-Sharing Analysis

```
Global State: $OPA$ - origin-sensitive pointer analysis,
    $visitedMethods \leftarrow \emptyset$, // flag visitedMethods
$m \leftarrow main$. // the main method
VisitMethod($m$)

VisitMethod($\Delta$, $y$):
$visitedMethods.add(m)$
foreach $s \in m.statements$ do
    switch $s$ do
        case $x.f$:                                   // read/write object field
            $origins \leftarrow$ FindPointsToOrigins($p$)
            foreach $\mathbb{O} \in origins$ do
                ComputeOriginSharing($s$,$f$,$\mathbb{O}$,$read/write$)
            end
        end
        case $x[idx]$:                                      // read/write array
            $origins \leftarrow$ FindPointsToOrigins($a$)
            foreach $\mathbb{O} \in origins$ do
                ComputeOriginSharing($s$,$*$,$\mathbb{O}$,$read/write$)
            end
        end
        case $m(args)$:                                    // call a new method
            foreach $m' \in$ FindCalleeMethods($m(args)$) do
                if $!visitedMethods.contains(m')$ then
                    VisitMethod($m'$)
                end
            end
        end
        otherwise do
            break
        end
    end
end

ComputeOriginSharing($s$, $f$, $\mathbb{O}$, $isWrite$):
Input      : $s$ - the statement;
             $f$ - accessed field ($*$ means array access);
             $\mathbb{O}$ - origin;
             $isWrite$ - true means write and false means read.
$WO \leftarrow$ GetWriteOrigins($s$,$f$)
$RO \leftarrow$ GetReadOrigins($s$,$f$)
if $isWrite$ && $!WO.contains(\mathbb{O})$ then
    $WO.add(\mathbb{O})$
end
if $!isWrite$ && $!RO.contains(\mathbb{O})$ then
    $RO.add(\mathbb{O})$
end
```

Table 6.3: The time complexity of different pointer analyses. Reprinted with permission from [4].

Analysis	Worst-Case Complexity
0-context	O$(p \times h^2)$
heap	O$(p^3 \times h^2)$
2-CFA + heap	O$(p^5 \times h^2)$
2-obj + heap	O$(p^5 \times h^2)$
1-origin + heap	O$(p^3 \times h^2)$

allocations, respectively. The complexity of k-CFA and k-obj varies according to the context depth k. However, their worst-case complexity can be doubly exponential [187]. The selective context-sensitive techniques [19, 79, 80, 81, 82, 130] are also bounded by the context depth and have the same worst-case complexity as their corresponding full k-CFA and k-obj algorithms.

The 1-origin has the same complexity as 1-call-site-sensitive heap analysis (denoted *heap*). But the number of operations is increased linearly by a factor $(\#\mathbb{O} \times \mathbb{O}\%)$, where $\#\mathbb{O}$ is the number of origins and $\mathbb{O}\%$ is the ratio between the average number of statements within an origin and the total number of program statements. The ratio is small (<10%) for most applications, according to our experiments in Section 6.4.

6.2.3 Origin-Sharing Analysis

Based on OPA, our origin-sharing analysis (OSA) uses an origin-sensitive heap abstraction and automatically identifies memory objects shared by different origins. A sample output is shown in Figure 6.2(d). A key in OSA is to track the objects accessed in the code path of each origin by leveraging OPA. Consistently with OPA, OSA is sound, interprocedural, and field-sensitive. More importantly, OSA is more scalable than conventional thread-escape analysis techniques [121, 122, 188] – it only requires a linear scan of the program statements.

As depicted in Algorithm 9, we traverse the program statements starting from the main entry method. There are three kinds of statements relevant to OSA:

- For each object field access (statement ❸❹ in Table 6.2), we query OPA to find all the possible allocated objects that the base reference may point to. Each object has an origin which is represented by its allocation site together with an origin. For each such origin $\mathbb{O}$, we call the procedure **ComputeOriginSharing**(s, f, $\mathbb{O}$, *isWrite*) to compute if the field access is shared by multiple origins or not. In ComputeOriginSharing, we maintain for each access a set of write origins and a set of read origins, retrieved by **GetReadOrigins** and **GetWriteOrigins**, respectively. If a field access in a statement is accessed by more than one origin, and with at least one of them is a write, we mark the access as *origin-shared*. For static field accesses, the procedure is similar except that each static field is directly encoded into a unique signature including the class name and the field index.

- For array accesses (statement ❺❻ in Table 6.2), we handle array accesses similar to that of object field accesses, but query about its field $*$ representing all array elements.

- For method invocation statements (statement ❼❽❾ in Table 6.2) (the receiver object is also included in the arguments `args`), we use OPA again to determine the possible callee methods and traverse their statements.

Compared to thread-escape analysis, OSA has the following key advantages:

- OSA is more general than thread-escape analysis since an origin can represent a thread or an event;

- OSA is more precise than thread-escape analysis. For example, static variables (and any object that is reachable from static variables) are often considered as thread-escaped. However, certain static variables may only be used by a single thread. OSA can distinguish such cases.

- While standard thread-escape analysis algorithms do not directly work for array accesses (because they have no information about array aliases), OSA can distinguish if an access $a[i]$ is an origin-shared array object or not through reasoning about the points-to set of $a.*$.

- OSA also identifies origin-shared reads and writes to provide fine-grained access information. This is particularly useful for static race detection and performance optimizations.

Table 6.4: SHB Graph with Origins: the following statements are in method `m()` with Origin $\mathbb{O}_i$. Reprinted with permission from [4].

Intra-Origin Happen-before Rules **Statement**	 **Intra-Origin Node & HB Edge**
❸ $x.f = y$	$\forall \langle o, \mathbb{O}_k \rangle \in pts(\langle x, \mathbb{O}_i \rangle)$, **write**$(\langle o, \mathbb{O}_k \rangle .f)$
❹ $x = y.f$	$\forall \langle o, \mathbb{O}_k \rangle \in pts(\langle y, \mathbb{O}_i \rangle)$, **read**$(\langle o, \mathbb{O}_k \rangle .f)$
❺ $x[idx] = y$	$\forall \langle o, \mathbb{O}_k \rangle \in pts(\langle x, \mathbb{O}_i \rangle)$, **write**$(\langle o, \mathbb{O}_k \rangle .*)$
❻ $x = y[idx]$	$\forall \langle o, \mathbb{O}_k \rangle \in pts(\langle y, \mathbb{O}_i \rangle)$, **read**$(\langle o, \mathbb{O}_k \rangle .*)$
❼ $x = y.f(a_1, ..., a_n)$	$\forall \langle f, \mathbb{O}_i \rangle \in dispatch(\langle y, \mathbb{O}_i \rangle, f)$, *add HB edge*: **call**$(\langle f, \mathbb{O}_i \rangle) \Rightarrow$ $\mathbf{f_{first}}(\langle f, \mathbb{O}_i \rangle)$, $\mathbf{f_{last}}(\langle f, \mathbb{O}_i \rangle) \Rightarrow$ $\mathbf{call_{next}}(\langle f, \mathbb{O}_i \rangle)$
🔓 $synchronized(x)\{$ $\ldots\}$	$\forall \langle o, \mathbb{O}_k \rangle \in pts(\langle x, \mathbb{O}_i \rangle)$, **lock**$(\langle o, \mathbb{O}_k \rangle)$, **unlock**$(\langle o, \mathbb{O}_k \rangle)$
Inter-Origin Happen-before Rules **Statement**	 **Inter-Origin Node & HB Edge**
❾ $x.entry(c_1, ..., c_n)$	$\forall \langle entry, \mathbb{O}_j \rangle \in dispatch(\langle x, \mathbb{O}_i \rangle, entry)$, *add HB edge*: **entry**$(\mathbb{O}_i, \mathbb{O}_j) \Rightarrow$ $\mathbf{origin_{first}}(\mathbb{O}_j)$
❿ $x.join()$	$\forall \langle join, \mathbb{O}_j \rangle \in dispatch(\langle x, \mathbb{O}_i \rangle, join)$, *add HB edge*: $\mathbf{origin_{last}}(\mathbb{O}_j) \Rightarrow$ **join**$(\mathbb{O}_j, \mathbb{O}_i)$

6.3 Static Data Race Detection

In O2, we model both threads and events statically as functional units, each represented by a static trace of memory accesses and synchronization operations. Our race detection engine uses hybrid happens-before and lockset analyses similar to most prior work on dynamic race detection [189] (although ours is static). More specifically, our detection represents happens-before relations by a static happens-before (SHB) graph [3, 27], which is designed to efficiently compute incremental changes from source code.

We modify the graph with origins as shown in Table 6.4. We record the field/array read and write accesses for statements ❸-❻ by creating read and write nodes. For statement ❼, we create a method call node (call) with two happens-before (HB) edges (denoted ⇒): one points from the call node to the first node ($\mathrm{f_{first}}$) of its target method `f` within the same origin $\mathbb{O}_i$, the other points from the last node (f_{last}) of $\langle f, \mathbb{O}_i \rangle$ to the next node after the call ($\text{call}_{\text{next}}$). Intra-origin HB edges are created by pointing from one intra-origin node to another in their statement order.

For lock operation 🔓, we create lock and unlock nodes to maintain the current lockset. For Java programs, we consider `synchronized` blocks and methods. For C/C++ programs, O2 currently only considers monitor-style locks (including both standard pthread mutexes and customized locks through configurations). And we aim to support atomics (*e.g.*, `std::atomic`) and semaphores in our future work, by adding new happens-before rules from different origins to the atomic/semaphore operations.

For calls to an origin entry point ❾, we create an origin entry node (entry) to represent the start of a new origin $\mathbb{O}_j$ from its parent origin $\mathbb{O}_i$. And we add an inter-origin HB edge pointing to the first node ($\text{origin}_{\text{first}}$) of $\mathbb{O}_j$. For thread join statement ❿, we create a join node (join) to indicate the end of $\mathbb{O}_j$ that finally joins to $\mathbb{O}_i$. An inter-origin HB edge is created from the last node ($\text{origin}_{\text{last}}$) of current origin ($\mathbb{O}_j$) to the join node.

Existing static race detection (such as [27]) typically checks each pair of two conflict accesses from different threads: run a depth-first search (or breadth-first search) starting from one access and vice versa to check their happens-before relation on the SHB graph, and compute the locksets for both accesses to check whether they have common lock guards.

However, the efficiency is limited by the redundant work in graph traversals and lockset retrievals for all pairs of memory accesses. The straw man approach cannot scale to real-world programs which can generate large SHB graphs with millions of memory accesses.

6.3.1 Three Sound Optimizations

To address the performance challenges, we develop the following sound optimizations:

Check Happens-Before Relation We only create inter-origin HB edges in the SHB graph. Instead of creating intra-origin HB edges, we assign a unique *integer ID* to each node, which is monotonically increased during the SHB construction. Therefore, we convert the traversal of visiting all intra-origin nodes along HB edges to a constant time integer comparison.

Check Lockset Intuitively, a list of locks is associated with each memory access node in the SHB graph in order to represent the mutex protection. We observe that the number of different combinations among mutexes is much smaller than the number of conflict memory accesses we need to check. Therefore, we assign each combination of mutexes (including the empty lockset) a *canonical ID* and associate each access node with such an ID. This not only reduces the memory for storing the SHB graph, but also speeds up the lockset checking process. All memory accesses with an identical lockset ID, or different IDs corresponding to overlapping locksets, are protected by the same lock(s), and the intersection of the IDs between two locksets can be cached for later checks.

Lock-Region-based Race Detection We observe that a synchronization block or method often guards a large sequence of memory accesses on the *same* origin-shared object(s) (o_s), which incurs redundant race checking. Instead, we treat all the memory accesses on o_s within the same lock region as a single memory access on o_s, and check races on that single access *once*. This is sound because their happens-before relations and locksets are exactly the same. This optimization significantly boosts O2's performance by reducing the number of memory access pairs for detecting data races.

6.3.2 Unify Threads with Android Events

Mobile applications are a representative class of modern software that contains complex interactions between threads and events. For instance, in Android apps, there are hundreds of different types of events that can be created from the Activity lifecycles, callbacks, UI, or the system services [190]. Meanwhile, the app logic may create any number of normal Java threads and AsyncTask to improve performance.

Keen readers may wonder that O2 may not work well for mobile apps, since such event-driven applications will generate a large number of origins. However, as we will show in our experiments, O2 scales well on Android apps, because Android apps often have short-duration events that explore only a small fraction starting from the entry points.

O2 detects data races in Android apps through the following treatments. In Android apps, there is no explicit main method as in other Java programs that can be used as the analysis entry of O2. Instead, we automatically generate an analysis harness from the main `Activity` of every Android app (*i.e.*, the home screen). The main activity can be identified by parsing the file *AndroidManifest.xml* within each Android apk.

Our tool treats each event handler as an origin entry. Once we hit a `startActivity()` or `startActivityForResult()`, we create a harness for the activity being started and analyze the new harness. All lifecycle event handlers are treated as method calls, while the normal event handlers are viewed as origin entries in OPA and SHB graph construction. Since all events are handled by the main thread [191], we protect the memory accesses within all the event handlers by one global lock, so that no false positive among event handlers will be reported by O2.

6.3.3 Other Implementation Details

Sequential and Relaxed Memory Models Different from sequential consistency, a relaxed memory model may reorder certain reads and writes in the same thread and different threads may see different orders. O2 works for both sequential and relaxed memory models. The reason is that the SHB graph captures inter-origin happens-before relations at synchronization sites, and it does not assume a global ordering of reads and writes. Hence, our happens-before relations already relax the ordering constraints for reads and writes from the same origin.

Cross-Module and External Pointers For C/C++, O2 always links the IR files into a single LLVM module and performs the analysis based on the whole module. Meanwhile, there is always a default origin (starting from the main entry point), so we do not have to deal with cross-module pointers. For JVM applications, O2 extends WALA's ZeroOneCFA to analyze all bytecode-level pointers loaded by the application classloader. When a pointer is passed from an external function call for which the IR file does not exist, we will create an anonymous object for that pointer.

Table 6.5: Performance comparison on JVM programs (in $sec.$). The left part compares OPA with other pointer analyses. The right part compares O2 with other race detection algorithms. The slowdown (in red) is normalized with 0-ctx as the baseline. Reprinted with permission from [4].

	Pointer Analysis							Race Detection						
App	**0-ctx**	**#O**	**OPA**	**1-CFA**	**2-CFA**	**1-obj**	**2-obj**	**0-ctx**	**O2**	**1-CFA**	**2-CFA**	**1-obj**	**2-obj**	**RacerD**
Avrora	13.42	4	15.56	17.81	56.06	50.30	>4h	20.70	17.85/**-14%**	30.63/0.48x	615.42/29x	1064/50x	-	18.36
Batik	7.22	4	9.83	49.28	2606	>4h	>4h	14.47	14.93/3%	84.01/4.81x	2648/182x	-	-	1min12s
Eclipse	5.03	4	6.52	8.43	8.71	11.84	>4h	7.21	8.03/11%	20.35/1.82x	40.36/4.60x	641.38/88x	-	*
H2	49.95	3	111.37	192.71	1397	>4h	>4h	58.13	169.63/192%	263.00/3.52x	3208/54x	-	-	38.25
Jython	25.77	4	66.34	16.09	58.85	>4h	>4h	163.49	537.63/229%	100.35/-0.39x	172.46/0.05x	-	-	1min47s
Luindex	10.23	3	15.73	16.74	26.38	>4h	>4h	14.43	20.19/40%	31.62/1.19x	1634/112x	-	-	2min39s
Lusearch	5.06	3	5.66	31.60	2384	6.48	6.63	7.09	7.99/13%	33.79/3.77x	2401/338x	8.58/0.21x	15.05/1.12x	2min39s
Pmd	5.50	3	5.75	6.04	8.00	>4h	>4h	25.07	13.32/**-47%**	57.21/1.28x	122.93/3.90x	-	-	2min15s
Sunflow	3.13	9	5.68	5.08	5.44	5.12	>4h	20.67	26.01/26%	297.05/13x	2408/116x	3007/144x	-	14.77
Tomcat	3.77	6	10.18	30.47	2312	5.89	676.59	7.58	16.87/123%	66.48/7.77x	2829/372x	2918/384x	9589/1.3kx	1min31s
Tradebeans	4.96	3	6.05	6.76	7.54	>4h	>4h	8.19	12.05/47%	16.49/1.01x	111.80/13x	-	-	9.38
Tradesoap	6.25	3	7.44	8.33	9.31	>4h	>4h	10.22	10.77/5%	24.32/1.38x	149.53/14x	-	-	9.38
Xalan	31.30	3	35.73	65.51	3922	213.99	>4h	34.71	42.87/24%	79.33/1.29x	5722/164x	3h/305x	-	32.87
ConnectBot	2.40	11	5.45	23.85	3513	>4h	>4h	2.49	5.57 /124%	23.99/8.63x	3513/1.4kx	-	-	*
Sipdroid	5.80	15	31.48	14.33	3436	>4h	>4h	16.02	228.33/1.3k%	40.88/1.55x	3452/215x	-	-	*
K-9 Mail	6.56	23	14.73	30.88	4284	>4h	>4h	8.59	19.49/127%	33.32/2.88x	4288/498x	-	-	4min56s
Tasks	6.90	7	12.72	117.63	8081	>4h	>4h	7.10	12.90/82%	117.77/15.59x	8081/1.1kx	-	-	*
FBReader	6.66	15	20.16	45.26	2.97h	>4h	>4h	7.49	23.33 /211%	52.79/6.05x	3.10h/1.5kx	-	-	*
VLC	5.35	4	46.40	25.40	3235	>4h	>4h	5.39	46.44/762%	25.44/3.72x	3235/599x	-	-	*
FireFox Focus	3.84	8	15.46	17.96	>4h	>4h	>4h	4.08	15.76/286%	18.34/3.50x	-	-	-	2min5s
Telegram	20.82	134	199.79	83.31	>4h	>4h	>4h	41.76	372.93/793%	171.42/3.10x	-	-	-	*
Zoom	36.77	15	148.01	198.59	>4h	>4h	>4h	37.62	149.01/296%	200.47/4.33x	-	-	-	*
Chrome	6.14	34	108.76	18.43	>4h	>4h	>4h	7.35	111.79/1.4k%	22.72/2.09x	-	-	-	*
HBase	41.96	16	494.64	61.75	>4h	>4h	>4h	>4h	1.34h/**-66.5%**	>4h	-	-	-	8min12s
HDFS	29.03	12	102.83	40.35	165.05	>4h	>4h	>4h	499.53/**-28x**	>4h	>4h	-	-	3min22s
Yarn	416.37	14	603.70	61.42	55.58	>4h	>4h	>4h	1.7h/**-57.5%**	>4h	>4h	-	-	8min5s
ZooKeeper	14.40	40	33.45	15.32	33.31	>4h	>4h	>4h	271.20/**-53x**	>4h	>4h	-	-	21.18

"#O": The number of origins detected during the analysis.
"-": Time out.
"*": RacerD could not run successfully due to compiler errors.

6.4 Experiments

We evaluated O2 on a large collection of real-world, widely-used distributed systems (*e.g.*, *ZooKeeper* and *HBase*), Android apps (*e.g.*, *Firefox* and *Telegram*), key-value stores (*e.g.*, *Redis/RedisGraph*, *Memcached* and *TDengine*), network controllers (*Open vSwitch OVS*), lock-free algorithms (*e.g.*, *cpqueue* and *mrlock*), as well as the *Linux kernel*.

6.4.1 Performance for Java, Android and C/C++

6.4.1.1 OPA vs Other Pointer Analyses

The left part of Table 6.5 summarizes the performance of different pointer analysis algorithms on the JVM benchmarks, including Dacapo [155], a collection of popular Android apps and dis-

Table 6.6: Performance comparison on C/C++ benchmarks (in *sec.*). The slowdown (*SD*) is normalized with 0-ctx as the baseline. Reprinted with permission from [4].

App	**#KLOC**	**Metrics**	**0-ctx**	**O2**	**2-CFA**
Memcached (#O = 12)	20.4	Time/SD	5.3	5.8/9%	7.5/41%
		#Pointer	8,400	12,883	15,772
		#Object	2,420	2,468	2,765
		#Edge	5,395	10,415	17,116
Redis (#O = 15)	116	Time/SD	9.3	15.0/61%	275.9/28x
		#Pointer	44,535	54,690	281,524
		#Object	14,458	14,913	32,401
		#Edge	598,981	963,654	13,530,084
Sqlite3 (#O = 3)	245	Time/SD	213	273/28%	OOM
		#Pointer	57,657	61,796	-
		#Object	10,093	10,310	-
		#Edge	7,909,626	8,879,155	-
Avg.	126	Time/SD	75.8	97.9/30%	-

tributed systems. Overall, OPA significantly outperforms 1-CFA, 2-CFA, 1-obj and 2-obj by 1x, 152x, 390x and 465x speedup (on average) respectively, and OPA has only a small performance slowdown compared to the context-insensitive baseline (denoted *0-ctx*, 1.76x on average). In particular, the majority of benchmarks running 1-obj and 2-obj cannot terminate within 4 hours.

Note that the number of origins (denoted *#O*) in the evaluated Android apps is significantly larger than that in the other JVM applications, up to over a hundred origins in *Telegram*. However, OPA is still highly efficient, finishing in a few minutes in the worst case. Compared to the other algorithms, the performance of origin-sensitivity is comparable to 1-CFA (but much more precise by identifying thread-/event-local points-to constraints) and several orders of magnitude faster than 2-CFA, 1-obj and 2-obj.

The scalability of k-obj [49] and k-CFA [126] varies depending on the code. For most benchmarks, more objects are allocated when running k-obj than k-CFA, *e.g.*, 2-CFA allocates 3357 objects for *Tomcat*, while 2-obj allocates 20679. Meanwhile, opposite cases exist, *e.g.*, *Avrora* has 7369 objects for 1-CFA and 5848 for 1-obj.

Table 6.7: Performance and #Shared memory accesses (#S-access) of OSA. Reprinted with permission from [4].

App	**#S-access**	**Time**
Avrora	16	16.72s
Batik	293	7.79s
Eclipse	343	9.22s
H2	2,207	2.3min
Jython	13,121	4.5min
Luindex	2,001	1.6min
Lusearch	252	7.01s
Pmd	300	8.57s
Sunflow	1,603	10.15s
Tomcat	700	15.39s
Tradebeans	45	7.43s
Tradesoap	37	9.12s
Xalan	14	1.2min

Time includes the time of OPA.

Table 6.6 reports the performance for three C/C++ applications (*Memcached*, *Redis* and *Sqlite3*). OPA achieves upto 17x speedup over 2-CFA on *Redis* while only incurring 30% slowdown compared with 0-ctx. Moreover, 2-CFA got killed when running on *Sqlite3* due to out of memory (OOM, 32GB) while OPA only imposes 28% slowdown. We note that O2 detected numerous real races in all these three applications. We will elaborate the case of *Memcached* in Section 6.4.4.

6.4.1.2 OSA vs Escape Analysis

We compared OSA with an open-source escape analysis TLOA [122], which is integrated in the state-of-the-art static analysis framework Soot [72]. TLOA uses context-sensitive information flow analysis to decide whether a field can be accessed by multiple threads. Table 6.7 reports the number of thread-shared accesses for each benchmark computed by OSA, which has the same setting with the evaluation of OPA. OSA completes in 51s on average, while TLOA could not finish within the time limit for all the benchmarks. We further excluded JDK libraries and the benchmark-specific dependencies (*e.g.*, *antlr* and *asm* for *Jython*). However, TLOA only finishes the analysis for *Avrora* in 90s, which generates an imprecise report with no thread-escape accesses.

Table 6.8: #Races detected by O2 and D4 utilizing different pointer analyses. The percentages of reduced races (in red) are normalized with 0-ctx as the baseline. The comparison between O2 and RacerD (v1.0.0) is shown separately on the right, due to different scale. Reprinted with permission from [4].

App	0-ctx	O2	1-CFA	2-CFA	1-obj	2-obj	O2	RacerD
Avrora	12,633	38/99.7%	45/99.6%	45/99.6%	47/99.6%	-	38	117
Batik	4,369	186/95.7%	4,229/3.2%	640/85.4%	-	-	186	1,562
Eclipse	958	7/99.3%	944/1.5%	822/14.2%	945/1.4%	-	7	*
H2	9,698	2,817/71.0%	7,832/19.2%	6,322/34.8%	-	-	2,817	6,743
Jython	7,997	3,651/54.3%	2,402/70.0%	2,358/70.5%	-	-	3,651	52,872
Luindex	3,218	1,792/44.3%	2,821/12.3%	2,271/29.4%	-	-	1,792	172
Lusearch	567	341/39.9%	538/5.1%	494/12.9%	529/6.7%	526/7.2%	341	172
Pmd	307	256/16.6%	296/3.6%	293/4.6%	-	-	256	1
Sunflow	9,238	1,925/79.2%	6,868/25.7%	5,899/36.1%	2,288/75.2%	-	1,925	69
Tomcat	751	307/59.1%	701/6.7%	693/7.7%	585/22.1%	575/23.4%	307	3,257
Tradebeans	193	75/61.1%	171/11.4%	168/13.0%	-	-	75	90
Tradesoap	264	64/75.8%	179/32.2%	177/33.0%	-	-	64	90
Xalan	6	1/83.3%	6/0.0%	6/0.0%	6/0.0%	-	1	754

"-": Time out.

For the other benchmarks, TLOA still cannot finish within one hour, which is over 70x (on average) slower than OSA.

6.4.1.3 Race Detection Performance

The right part of Table 6.5 reports the performance for race detection including the time of running the corresponding pointer analysis. In summary, O2 achieves 70x speedup on average over the other context-sensitive detections. Among them, the most speedup (1461x on detection and 568x in total) is on *Tomcat* when comparing with 2-obj. Compared to 0-ctx, O2 is only 2.81x slower on average, and it is even faster for some applications (*Avrora* and *Pmd*), due to the much improved precision of origin-shared memory accesses.

O2 vs RacerD We also compared O2 with RacerD (from the latest release Infer v1.0.0) in Table 6.5. RacerD did not complete the detection for 9 out of the 27 benchmarks, due to dependency limitation of the benchmark or compilation errors. For example, *Eclipse* has a complex building procedure, *Sipdroid* requires Android command which RacerD does not support, and other Android benchmarks involve legency SDK that could not be resolved. O2 (69s on average) and

Table 6.9: #Races and #Thread-shared objects (#S-obj) from different pointer analyses for distributed systems. Reprinted with permission from [4].

App	#Race O2	#Race RacerD	#S-obj 0-ctx	#S-obj 1-CFA	#S-obj 2-CFA	#S-obj O2
HBase	687	727	1,269	1,799	-	903
HDFS	910	884	2,322	3,139	6,605	1,066
Yarn	1,164	1,246	5,387	3,083	2,146	1,162
ZooKeeper	747	407	1,389	2,511	4,299	1,271

"-": Time out.

Table 6.10: New Races Detected by O2 (Confirmed by Developers). Reprinted with permission from [4].

	Linux	TDengine	Redis/RedisGraph	OVS	cpqueue	mrlock	Memcached	Firefox	ZooKeeper	HBase	Tomcat
#Races	6	6	5	3	7	5	3	2	1	1	1

RacerD (71s on average) have similar performance on Dacapo benchmarks, while RacerD is 90% slower on average on the two Android benchmarks (*i.e.*, *K-9 Mail* and *Firefox Focus*). For the four distributed systems, O2 (48min, including the execution of a whole program pointer analysis), has 9.6x slowdown on average comparing with RacerD (5min). We also tested RacerD on the three C/C++ programs in Table 6.6. However, RacerD could not run successfully on *Memcached* and *Redis*, and it reports no violations on *Sqlite3*.

6.4.2 Precision of Origin-Sensitivity and O2

Tables 6.8 and 6.9 report the number of detected races with different pointer analyses. We use the number of reported races as the metric to evaluate the end-to-end precision of different analyses. The baseline is the open-source tool developed in D4 [3], which utilizes the points-to result from 0-ctx for static race detection.

In summary, O2 reduces warnings by 77% on average, while 1- and 2-CFA reduce 46% and 60%, and 1- and 2-obj reduce 35% and 19% respectively. For the majority of benchmarks, O2 reports significantly fewer races, *e.g.*, *Eclipse*. For other benchmarks, O2 is much faster to achieve a similar precision. For example, O2 reports 38 races for *Avrora*, both 1- and 2-CFA report 45 races, and 1-obj reports 47 races (while being 60x slower than O2).

O2 vs RacerD RacerD reports two types of thread safety violations in the evaluated benchmarks: (1) read/write races, and (2) unprotected write violations where a field access at a program location is outside of synchronization. To perform a fair comparison, we translate the violations in the RacerD report to a number of potential races: we add up the numbers of read/write races and of the pairs of conflict field accesses shown in unprotected writes. For distributed systems, we report the detected races in Table 6.9, since all the other context-sensitive detections run out of time (>4h).

On average, O2 reports 4.33x fewer warnings compared to RacerD (reduces false positives by 82% from Table 6.8). O2 detects new races in *FireFox Focus, TDengine, OVS, Memcached* and *Redis*; while RacerD either reports no races or cannot complete its detection on those programs. The majority of false positives reported by O2 are due to infeasible paths, which is inherent to static analysis tools.

6.4.3 Trade-Off between Precision and Performance

Our results show that O2 significantly outperforms k-CFA and k-obj ($k \leq 2$) in terms of both performance and precision (Table 6.5). The reason for the improved precision is that the use of origins as context significantly improves the analysis precision on the thread- and event-local objects that are created within an origin. Such origin-local objects would be falsely analyzed as shared by k-CFA or k-obj if *k* is smaller than the depth of the call chain inside the thread or event, whereas such objects can be correctly analyzed as thread-local by O2.

The performance of OPA has obvious slowdown on the distributed systems: the max slowdown is 9.86x on *Yarn* compared with 2-CFA. However, its corresponding total time of race detection is at least 57% faster (up to 53x on *ZooKeeper*). The reason behind this significant speedup is the largely reduced number of thread-shared objects as shown in Table 6.9, which means less workload in both checking happens-before relations and computing common locksets.

6.4.4 New Races Found in Real-World Software

O2 has detected new races in every real-world code base we tested on, as summarized (partially) in Table 6.10. Most of them are due to a combination of threads and events. If considering events only or threads only, or considering them separately, these races will be missed. In the following, we elaborate the races found in several high-profile C/C++, Android apps, and distributed systems.

Linux Kernel We evaluated O2 on the Linux kernel (commit 5b8b9d0c as of April 10th, 2020), compiled with tinyconfig64, clang/LLVM 9.0. We define four types of origins: system calls with function prefix: __x64_sys_xx, driver functions over file operations (owner, llseek, read, write, open, release, etc), kernel threads with origin entries kthread_create_on_cpu() and kthread_create_on_node(), and interrupt handlers with origin entries request_threaded_irq() and request_irq()). There are 398 system calls included in our build. For each system call, we create two origins representing concurrent calls of the same system call, and a shared data pointer if the system call has a parameter that is a pointer (*e.g.*, __x64_sys_mincore). In total, 1090 origins are created, including 796 from system calls and 294 from others.

In total, O2 detects 26 races in less than 8 minutes. We manually inspected all these races and confirmed that 6 are real races, 7 are potential races, and the other 13 are false positives. The 6 real races are all races to the linux kernel bugzilla, and all of them have been confirmed at the time of writing. The 7 potential races are difficult to manaully inspect due to very complex code paths involving the races. For the false positives, a majority of them are due to mis-recognition of spinlocks (such as arch_local_irq_save.38) or infeasible branch conditions which O2 does not handle. The code snippet below shows a real bug found by O2, which detects concurrent writes on the same element of array *vdata* (with array index CS_HRES_COARSE).

```
void update_vsyscall_tz(void){ //in class time.vsyscall
  struct vdso_data *vdata = __arch_get_k_vdso_data();
  vdata[CS_HRES_COARSE].tz_minuteswest = sys_tz.tz_minuteswest; //RACE
  vdata[CS_HRES_COARSE].tz_dsttime = sys_tz.tz_dsttime; //RACE
  ...
}
```

In addition, we found that among the 71459 allocated objects by the kernel (within the configured origins), 329 of them are origin-shared. And 1051 accesses are on origin-shared memory locations from a total of 36321 memory accesses. The result indicates that the majority of memory used by the kernel is origin-local, which can be beneficial to region-based memory management.

We also discovered that the system call paths do not create any new kernel threads or register interrupts. However, driver functions can do both operations. For example, the driver of GPIO requests a thread to read the events by the kernel API `request_threaded_irq` [†]. And the interrupt requests can create kernel threads by API `kthread_create` [‡].

Memcached Memcached is a high performance multithreaded event-based key/value cache store widely used in distributed systems. We applied O2 to commit `14521bd8` (as of May 12th, 2020). O2 is able to finish analyzing memcached within 5s, and reports 16 new races in total. All these races are previously unknown. We manually confirmed that 11 of them are real and the rest of them are potential races. A majority of the real races are on variables such as `stats`, `settings`, `time_out`, or `stop_main_loop`. There are also three races that are not on these variables but look more harmful. We reported the three races to the developers and all of them have been confirmed. The other five potential races all involve pointer aliases on queued items.

One of the reported races is shown below with the simplified code snippet:

```
void *do_slabs_reassign(){ //event
  ...
  if (slabsclass[id].slabs > 1){
    return cur;//RACE: missing lock
  }
}
void *do_slabs_newslabs(){ //thread
  ...
  pthread_lock();
  p->slab_list[p->slabs++] = ptr;//with lock
  pthread_unlock();
  ...
}
```

The listed bug is related to Memcached's *slab-base memory allocation*, which is used to avoid

[†]/linux-stable/drivers/gpio/gpiolib.c@1104:8
[‡]/linux-stable/kernel/irq/manage.c@1279:7 and @1282:7

memory fragmentation by storing different objects using different *slab classes* based on their size. Since the accesses in the event handler is not protected by the lock, there is a data race between the event handler and all the running threads that try to allocate new slabs. Although another lock-protected check on the same variable is made later in the function, the data race can still lead to undefined behaviors. This case is interesting as it shows that unlike previous tools, which only reason about *inter-thread* races, O2 is able to unify events and threads to find races in complex programs that leverage both concepts for concurrency.

FireFox Focus O2 was able to finish in 15s on FireFox Focus 8.0.15 (a privacy-focused mobile browser), and detected two previously unknown bugs (both reported in Bug-1581940) confirmed by developers from Mozilla. A simplified code snippet is presented below:

```
// called from Gecko background thread
public synchronized IChildProcess bind(){
  ...
  Context ctx = GeckoAppShell.getAppCtx();//RACE
  ...
}
// called from MainActivity.onCreate()
@UiThread
public void attachTo(Context context){
  ...
  Context appCtx = context.getAppCtx();
  if(!appCtx.equals(GeckoAppShell.getAppCtx())){
    GeckoAppShell.setAppCtx(appCtx);//RACE
  }
}
```

The code involves both FireFox Focus and FireFox's browser engine, Gecko. Upon the app initialization, `GeckoAppShell.getAppCtx()` and `GeckoAppShell.setAppCtx(appCtx)` are called without synchronizations, one from Android UI thread (through `onCreate` event handler), the other from Gecko engine's background thread. Although in reality, the creation order between UI thread and Gecko background thread keeps the race from happening, it is possible for Gecko engine to read an uninitialized application context thus leads to crash.

Distributed Systems We discovered two new races in ZooKeeper 3.5.4 (reported in ZOOKEEPER-3819) and HBase 2.8.0 (reported in HBase-24374). O2 takes 4.5min to detect the new race in ZooKeeper by analyzing 40 threads and 88 events. The related code is shown below:

```
1 //in class org.apache.zookeeper.server.DataTree
2 public void createNode(..., long ephemeralOwner){ ...
3   HashSet<String> list =ephemerals.get(ephemeralOwner);
4   if (list == null){
5     list = new HashSet<String>();
6     ephemerals.put(ephemeralOwner, list);
7   }
8   synchronized (list) {//RACE
9     list.add(path);
10  }
11  ...
12 }
13 public void deserialize(InputArchive ia, String tag){
14   HashSet<String> list = ephemerals.get(eowner);
15   if (list == null){
16     list = new HashSet<String>();
17     ephemerals.put(eowner, list);
18   }
19   list.add(path);//RACE: missing lock
20 }
```

These races are caused by interactions between threads and requests. *ephemerals* is a map in class `DataTree` to store the paths of the ephemeral nodes of a session. It is possible that a request of creating nodes for a session might arrive together with another request to deserialize the same session, and both requests are handled by different server threads (with super type *ZooKeeperServer*). The lock protection is missing on variable *list* on line 22, hence both threads can add paths concurrently to *ephemerals*. A worse case is that the two code snippets (line 4-7 and line 10-13) are not protected by common locks or mechanism from *ConcurrentHashMap*. Hence, the null checks from two threads on variable *list* may return null, but only one initialized set can be stored in *ephemerals* and all the paths added by another thread are missing. The race in *HBase* has the same reason as above, involving two concurrent accesses on a map, *keyProviderCache*, without locks from method `getKeyProvider()` (in class `org.apache.hadoop.hbase.io.crypto.Encryption`).

6.5 Summary

We have presented O2, a new system for static race detection. O2 is powered by a novel abstraction, *origins*, that unifies threads and events to effectively reason about shared memory and pointer aliases. Our extensive evaluation with Java and C/C++ programs demonstrates the potential of O2, finding a large number of new races in mature open-source code bases and achieving dramatic performance speedups and precision improvement over existing static analysis tools. O2 has been integrated into Coderrect, a commerical static analyzer [192]. In future work, we plan to implement and evaluate O2 for other languages such as Golang, C# and Rust.

7. CONCLUSION AND FUTURE WORK

This thesis makes contributions to incremental algorithms of pointer analysis and static concurrency bug detection.

Along the direction of pointer analysis, we present two sets of incremental algorithms, IPA and SHARP, that are efficient and scalable to real-world Java programs. Established on the new properties we observed on acyclic PAGs, we present IPA, an end-to-end incremental algorithm for context-insensitive on-the-fly pointer analysis. IPA handles both statement addition and deletion that affects the result of pointer analysis, and maintains the soundness and precision as the result of running whole program pointer analysis. The major contribution of IPA is achieving efficiency when handling deletions by avoiding the redundant computation and expensive graph reachability check from existing incremental algorithms. Moreover, we parallel IPA to further boost the performance. To demonstrate the efficiency, we evaluate IPA on DaCapo benchmarks which is 200X faster than existing state-of-the-art algorithms, while our parallel version only require 24ms on average.

In order to obtain more precise points-to results, this thesis presents SHARP, an incremental algorithm for context-sensitive pointer analysis that can be generalized to *k-CFA* and *k-obj*. Instead of applying IPA on context-sensitive PAG and CG naively, we identify redundant computation that iteratively discovers invalid nodes and edges in CG and PAG for a deleted method call statement. To avoid such redundancy, we propose a precompute algorithm to identify the invalid graph elements before conducting any update on CG and PAG. Besides, we discuss more parallel scenarios according to real-world GitHub commits with respect to efficiency, redundancy and conflict. We conduct an empirical evaluation on real-world GitHub projects. SHARP only requires on average 31s to handle a real-world code commit for *k-CFA* and *k-obj*, and our parallelization further improves the performance to 18s per code commit on average on an eight-core machine. Moreover, the algorithms in IPA and SHARP can be applied to other programming languages, *e.g.*, C/C++ and Golang.

Along the direction of static concurrency bug detection, this thesis proposes an static analysis framework, D4, to detect data races and deadlocks in multithreaded Java programs during programming phase. Aided by IPA, we design three efficient incremental algorithms on happens-before analysis, lock-dependency analysis and static concurrency bug detection, in order to achieve instant feedback. We also evaluate D4 on DaCapo benchmarks and compare with the state-of-the-art incremental concurrency bug detection tools, ECHO. D4 achieves 10X-2000X speedup, which is at least 2000X faster than the whole program pointer analysis.

To improve the precision of static concurrency bug detection, we propose "origin" to unify the concepts of thread and event by treating them as entry point and their attributes. We present origin-sensitive pointer analysis that precisely and efficiently distinguish origin-shared and origin-local objects, and show this result in our origin-sharing analysis. We leverage the result of origin-sensitive pointer analysis to detect concurrency bugs in multithreaded and event-driven programs. To demonstrate the efficiency and precision of origins, we compare our origin-sharing analysis with TLOA, a thread escape analysis from Soot. Our analysis only requires 51s on average on DaCapo benchmarks, while TLOA cannot terminate within an hour. We also evaluate our origin-based concurrency bug detector on many Java, Android programs and distributed systems, and compared with the commercial tool, RacerD. Our detector achieves the similar performance as RacerD, but reports 4.33X fewer races.

Future Work In the future, I would like to pursue two main directions in order to radically improve software quality as well as the experience of developers. The first direction is to collaborate static analysis with dynamic techniques, in order to improve the precision and confidence of the detection results. And the second direction is to apply and design static techniques for detecting vulnerabilities in new and popular programs and languages.

Static error detection techniques still have a long way to go before achieving a reliable, universal tool to help developers improve the quality of code for everyday usage. Users often lack of the confidence in speed and precision of static tools due to the hard-to-scale nature and the mixing of true and false positives. The work D4 proves the potential of static analysis tools in the respect of performance. Next step is to improve its precision by verifying reported bugs with concrete input and execution traces. To reach this goal, dynamic tools, which only report true positive, is the most useful assistant. Concolic testing, as a practical example of combining static symbolic execution with concrete runs, inspires me a lot in integrating static concurrency bug detection techniques with efficient dynamic analyses (*e.g.*, fuzzing). Existing hybrid race detection techniques [181, 193, 194, 195, 196] leverage static analysis to speed up their dynamic procedures by reducing the suspicious code area, however, cannot guarantee to discover all bugs in the program. I will explore the reverse way that dynamic techniques become efficient assistants to filter out false positives and even verify races reported by static analysis.

More new programming languages emerge according to the demands of new usage scenarios. Classic programming languages (Java, C/C++), widely-used and maturely-researched, can provide the precious lessons for the new ones. My goal is to apply my understanding and knowledge from the classic, mature languages to new, prominent languages. Languages with same features (*e.g.*, multithreaded, mutex) may expose to the same issues and vulnerabilities, but under different syntax and semantics with various seriousness. Moreover, the compilers (or interpreters), as the fundamental component of a language, share similar procedures and algorithms. This generality across languages triggers me to reconsider my works from a broader view, which in turn stimulates sound and practical program analysis tools to help more developers with their everyday coding.

REFERENCES

[1] B. Liu, J. Huang, and L. Rauchwerger, "Rethinking incremental and parallel pointer analysis," *ACM Trans. Program. Lang. Syst.*, vol. 41, no. 1, pp. 6:1–6:31, 2019.

[2] B. Liu and J. Huang, "Sharp: Fast incremental context-sensitive pointer analysis for java," vol. 6, apr 2022.

[3] 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, (New York, NY, USA), pp. 359–373, ACM, 2018.

[4] B. Liu, P. Liu, Y. Li, C.-C. Tsai, D. Da Silva, and J. Huang, "When threads meet events: Efficient and precise static race detection with origins," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, (New York, NY, USA), p. 725–739, Association for Computing Machinery, 2021.

[5] "Average number of new iOS app releases per month as of June 2022." `https://www.statista.com/statistics/1020964/apple-app-store-app-releases-worldwide`, 2022.

[6] "Top Google Play Store Statistics 2022 You Must Know." `https://appinventiv.com/blog/google-play-store-statistics`, 2022.

[7] "Development - Linux kernel." `https://en.wikipedia.org/wiki/Linux_kernel`, 2023.

[8] "Ibm gives cancer-killing drug ai project to the open source community." `https://www.zdnet.com/article/ibm-reveals-ai-projects-aiming-to-find-cancer-killing-drugs/`, 2022.

[9] “$280 million stolen per month from crypto transactions.” `https://cybernews.com/crypto/flash-boys-2-0-front-runners-draining-280-million-per-month-from-crypto-transactions/`, 2022.

[10] “Report: Software bug led to death in uber’s self-driving crash.” `https://arstechnica.com/tech-policy/2018/05/report-software-bug-led-to-death-in-ubers-self-driving-crash/`, 2022.

[11] R. Ghiya and L. J. Hendren, “Putting pointer analysis to work,” in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’98, (New York, NY, USA), p. 121–133, Association for Computing Machinery, 1998.

[12] R. A. Chowdhury, P. Djeu, B. Cahoon, J. H. Burrill, and K. S. McKinley, “The limits of alias analysis for scalar optimizations,” in *Compiler Construction* (E. Duesterwald, ed.), (Berlin, Heidelberg), pp. 24–38, Springer Berlin Heidelberg, 2004.

[13] M. A. El-Zawawy, “Program optimization based pointer analysis and live stack-heap analysis,” *CoRR*, vol. abs/1104.0644, 2011.

[14] P. Wu, P. Feautrier, D. Padua, and Z. Sura, “Instance-wise points-to analysis for loop-based dependence testing,” in *Proceedings of the 16th International Conference on Supercomputing*, ICS ’02, (New York, NY, USA), p. 262–273, Association for Computing Machinery, 2002.

[15] Y.-S. Hwang, “Parallelizing graph construction operations in programs with cyclic graphs,” *Parallel Computing*, vol. 28, no. 9, pp. 1307–1328, 2002.

[16] R. Ghiya, L. J. Hendren, and Y. Zhu, “Detecting parallelism in c programs with recursive data structures,” in *Compiler Construction* (K. Koskimies, ed.), (Berlin, Heidelberg), pp. 159–173, Springer Berlin Heidelberg, 1998.

[17] Y. Sui, X. Fan, H. Zhou, and J. Xue, “Loop-oriented pointer analysis for automatic simd vectorization,” *ACM Trans. Embed. Comput. Syst.*, vol. 17, jan 2018.

[18] B. Scholz, J. Blieberger, and T. Fahringer, "Symbolic pointer analysis for detecting memory leaks," in *Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, PEPM '00, (New York, NY, USA), p. 104–113, Association for Computing Machinery, 1999.

[19] S. Z. Guyer and C. Lin, "Error checking with client-driven pointer analysis," *Sci. Comput. Program.*, vol. 58, pp. 83–114, Oct. 2005.

[20] N. Dor, M. Rodeh, and M. Sagiv, "Detecting memory errors via static pointer analysis (preliminary experience)," in *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '98, (New York, NY, USA), p. 27–34, Association for Computing Machinery, 1998.

[21] 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), p. 157–166, Association for Computing Machinery, 1993.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] S. Zhan and J. Huang, "Echo: Instantaneous in situ race detection in the ide," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, (New York, NY, USA), p. 775–786, Association for Computing Machinery, 2016.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

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

[41] B. Steensgaard, "Points-to analysis in almost linear time," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, (New York, NY, USA), pp. 32–41, ACM, 1996.

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

[43] M. Sridharan, S. Chandra, J. Dolby, S. J. Fink, and E. Yahav, "Aliasing in object-oriented programming," ch. Alias Analysis for Object-oriented Programs, pp. 196–232, Berlin, Heidelberg: Springer-Verlag, 2013.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

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

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] 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), p. 243–262, Association for Computing Machinery, 2009.

[66] 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.

[67] 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.

[68] Y. A. Liu and S. D. Stoller, "From datalog rules to efficient programs with time and space guarantees," in *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming*, PPDP '03, (New York, NY, USA), pp. 172–183, ACM, 2003.

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

[70] 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.

[71] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn, "Design and implementation of the logicblox system," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, (New York, NY, USA), pp. 1371–1382, ACM, 2015.

[72] "Sable/soot: Soot - A Java optimization framework." `https://github.com/Sable/soot`, 2019.

[73] 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.

[74] G. Varró and F. Deckwerth, "A rete network construction algorithm for incremental pattern matching," in *Theory and Practice of Model Transformations* (K. Duddy and G. Kappel, eds.), (Berlin, Heidelberg), pp. 125–140, Springer Berlin Heidelberg, 2013.

[75] 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.

[76] J. Krainz and M. Philippsen, "Diff graphs for a fast incremental pointer analysis," in *Proceedings of the 12th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ICOOOLPS'17, (New York, NY, USA), Association for Computing Machinery, 2017.

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

[78] 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.

[79] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis, "Precision-guided context sensitivity for pointer analysis," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, p. 141, 2018.

[80] Y. Li, T. Tan, A. Møler, and Y. Smaragdakis, "Scalability-first pointer analysis with self-tuning context-sensitivity," in *Proc. 12th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, November 2018.

[81] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras, “Introspective analysis: context-sensitivity, across the board,” in *ACM SIGPLAN Notices*, vol. 49, pp. 485–495, ACM, 2014.

[82] S. Z. Guyer and C. Lin, “Client-driven pointer analysis,” in *Proceedings of the 10th International Conference on Static Analysis*, SAS’03, (Berlin, Heidelberg), pp. 214–236, Springer-Verlag, 2003.

[83] B. Hassanshahi, R. K. Ramesh, P. Krishnan, B. Scholz, and Y. Lu, “An efficient tunable selective points-to analysis for large codebases,” in *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2017, (New York, NY, USA), pp. 13–18, ACM, 2017.

[84] S. Jeong, M. Jeon, S. Cha, and H. Oh, “Data-driven context-sensitivity for points-to analysis,” *Proc. ACM Program. Lang.*, vol. 1, pp. 100:1–100:28, Oct. 2017.

[85] S. Wei and B. G. Ryder, “Adaptive context-sensitive analysis for javascript,” in *29th European Conference on Object-Oriented Programming (ECOOP 2015)* (J. T. Boyland, ed.), vol. 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 712–734, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015.

[86] M. Jeon, S. Jeong, and H. Oh, “Precise and scalable points-to analysis via data-driven context tunneling,” *Proc. ACM Program. Lang.*, vol. 2, pp. 140:1–140:29, Oct. 2018.

[87] 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.

[88] 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.

[89] “Galois System.” `http://iss.ices.utexas.edu/`, 2017.

[90] 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.

[91] 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.

[92] 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.

[93] 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.

[94] 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.

[95] M. G. Burke, P. R. Carini, J. Choi, and M. Hind, "Flow-insensitive interprocedural alias analysis in the presence of pointers," in *Languages and Compilers for Parallel Computing, 7th International Workshop, LCPC'94, Ithaca, NY, USA, August 8-10, 1994, Proceedings*, pp. 234–250, 1994.

[96] 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.

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

[98] 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.

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

[100] 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.

[101] 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.

[102] 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.

[103] H. C. Lauer and R. M. Needham, “On the duality of operating system structures,” *SIGOPS Oper. Syst. Rev.*, vol. 13, pp. 3–19, Apr. 1979.

[104] J. OUSTERHOUT, “Why threads are a bad idea (for most purposes),” 1995.

[105] E. A. Lee, “The problem with threads,” *Computer*, vol. 39, pp. 33–42, May 2006.

[106] R. von Behren, J. Condit, and E. Brewer, “Why events are a bad idea (for high-concurrency servers),” in *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, HOTOS’03, (Berkeley, CA, USA), pp. 4–4, USENIX Association, 2003.

[107] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer, “Capriccio: Scalable threads for internet services,” in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP ’03, (New York, NY, USA), pp. 268–281, ACM, 2003.

[108] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur, “Cooperative task management without manual stack management,” in *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC ’02, (Berkeley, CA, USA), pp. 289–302, USENIX Association, 2002.

[109] P. Haller and M. Odersky, “Actors that unify threads and events,” in *Proceedings of the 9th International Conference on Coordination Models and Languages*, COORDINATION’07, (Berlin, Heidelberg), pp. 171–190, Springer-Verlag, 2007.

[110] P. Li and S. Zdancewic, “A language-based approach to unifying events and threads,” 2006.

[111] R. Jhala and R. Majumdar, “Interprocedural analysis of asynchronous programs,” in *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’07, (New York, NY, USA), p. 339–350, Association for Computing Machinery, 2007.

[112] F. Wei, S. Roy, X. Ou, and Robby, “Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps,” *ACM Trans. Priv. Secur.*, vol. 21, Apr. 2018.

[113] M.-H. Yee, A. Badouraly, O. Lhoták, F. Tip, and J. Vitek, “Precise dataflow analysis of event-driven applications,” *arXiv preprint arXiv:1910.12935*, 2019.

[114] T. Kamph, “An interprocedural points - to analysis for event-driven programs,” diplomarbeit, TU Hamburg-Harburg, Jan. 2012.

[115] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, (New York, NY, USA), p. 259–269, Association for Computing Machinery, 2014.

[116] M. Madsen, F. Tip, and O. Lhoták, "Static analysis of event-driven node.js javascript applications," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, (New York, NY, USA), p. 505–519, Association for Computing Machinery, 2015.

[117] J. Qian and B. Xu, "Thread-sensitive pointer analysis for inter-thread dataflow detection," in *11th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS'07)*, pp. 157–163, March 2007.

[118] B.-M. Chang and J.-D. Choi, "Thread-sensitive points-to analysis for multithreaded java programs," in *Computer and Information Sciences - ISCIS 2004* (C. Aykanat, T. Dayar, and İ. Körpeoğlu, eds.), (Berlin, Heidelberg), pp. 945–954, Springer Berlin Heidelberg, 2004.

[119] Y. Sui, P. Di, and J. Xue, "Sparse flow-sensitive pointer analysis for multithreaded programs," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, (New York, NY, USA), p. 160–170, Association for Computing Machinery, 2016.

[120] A. Rountev, A. Milanova, and B. G. Ryder, "Points-to analysis for java using annotated constraints," in *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, (New York, NY, USA), p. 43–55, Association for Computing Machinery, 2001.

[121] A. Salcianu and M. Rinard, "Pointer and escape analysis for multithreaded programs," in *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, PPoPP '01, (New York, NY, USA), pp. 12–23, ACM, 2001.

[122] R. L. Halpert, C. J. F. Pickett, and C. Verbrugge, “Component-based lock allocation,” in *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pp. 353–364, Sep. 2007.

[123] E. Ruf, “Effective synchronization removal for java,” in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI ’00, (New York, NY, USA), pp. 208–218, ACM, 2000.

[124] D. Grunwald and H. Srinivasan, “Data flow equations for explicitly parallel programs,” in *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP ’93, (New York, NY, USA), pp. 159–168, ACM, 1993.

[125] R. Rugina and M. C. Rinard, “Pointer analysis for structured parallel programs,” *ACM Trans. Program. Lang. Syst.*, vol. 25, pp. 70–116, Jan. 2003.

[126] O. Shivers, “Control-flow analysis of higher-order languages,” tech. rep., 1991.

[127] M. Sharir, A. Pnueli, *et al.*, *Two approaches to interprocedural data flow analysis*, ch. 8, p. 189–233. New York University. Courant Institute of Mathematical Sciences, 1978.

[128] Y. Smaragdakis, M. Bravenboer, and O. Lhoták, “Pick your contexts well: Understanding object-sensitivity,” in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’11, (New York, NY, USA), pp. 17–30, ACM, 2011.

[129] T. Tan, Y. Li, and J. Xue, “Making k-object-sensitive pointer analysis more precise with still k-limiting,” in *Static Analysis* (X. Rival, ed.), (Berlin, Heidelberg), pp. 489–510, Springer Berlin Heidelberg, 2016.

[130] J. Lu and J. Xue, “Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity,” *Proc. ACM Program. Lang.*, vol. 3, Oct. 2019.

[131] M. Jeon, S. Jeong, and H. Oh, “Precise and scalable points-to analysis via data-driven context tunneling,” *Proc. ACM Program. Lang.*, vol. 2, Oct. 2018.

[132] L. N. Q. Do, K. Ali, B. Livshits, E. Bodden, J. Smith, and E. Murphy-Hill, "Just-in-time static analysis," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, (New York, NY, USA), pp. 307–317, ACM, 2017.

[133] "Infer : Racerd." `http://fbinfer.com/docs/racerd.html`, 2021.

[134] P. Zhou, R. Teodorescu, and Y. Zhou, "Hard: Hardware-assisted lockset-based race detection," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, (Washington, DC, USA), pp. 121–132, IEEE Computer Society, 2007.

[135] P. Fonseca, C. Li, and R. Rodrigues, "Finding complex concurrency bugs in large multi-threaded applications," in *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, (New York, NY, USA), pp. 215–228, ACM, 2011.

[136] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps, "Conseq: Detecting concurrency bugs through sequential errors," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, (New York, NY, USA), pp. 251–264, ACM, 2011.

[137] "CIL - Infrastructure for C Program Analysis and Transformation." `https://people.eecs.berkeley.edu/~necula/cil/`, 2016.

[138] S. Blackshear, N. Gorogiannis, P. W. O'Hearn, and I. Sergey, "Racerd: Compositional static race detection," *Proc. ACM Program. Lang.*, vol. 2, Oct. 2018.

[139] C.-H. Hsiao, S. Narayanasamy, E. M. I. Khan, C. L. Pereira, and G. A. Pokam, "Asyncclock: Scalable inference of asynchronous event causality," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, (New York, NY, USA), p. 193–205, Association for Computing Machinery, 2017.

[140] C.-H. Hsiao, J. Yu, S. Narayanasamy, Z. Kong, C. L. Pereira, G. A. Pokam, P. M. Chen, and J. Flinn, “Race detection for event-driven mobile applications,” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’14, (New York, NY, USA), p. 326–336, Association for Computing Machinery, 2014.

[141] P. Maiya, A. Kanade, and R. Majumdar, “Race detection for android applications,” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’14, (New York, NY, USA), p. 316–325, Association for Computing Machinery, 2014.

[142] V. Raychev, M. Vechev, and M. Sridharan, “Effective race detection for event-driven programs,” in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA ’13, (New York, NY, USA), p. 151–166, Association for Computing Machinery, 2013.

[143] Y. Hu and I. Neamtiu, “Static detection of event-based races in android apps,” in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’18, (New York, NY, USA), p. 257–270, Association for Computing Machinery, 2018.

[144] V. Kahlon, N. Sinha, E. Kruus, and Y. Zhang, “Static data race detection for concurrent programs with asynchronous calls,” in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE ’09, (New York, NY, USA), p. 13–22, Association for Computing Machinery, 2009.

[145] G. Safi, A. Shahbazian, W. G. J. Halfond, and N. Medvidovic, “Detecting event anomalies in event-based systems,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, (New York, NY, USA), p. 25–37, Association for Computing Machinery, 2015.

[146] X. Fu, D. Lee, and C. Jung, “nadroid: Statically detecting ordering violations in android applications,” in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, (New York, NY, USA), p. 62–74, Association for Computing Machinery, 2018.

[147] M. Hind, M. Burke, P. Carini, and J.-D. Choi, “Interprocedural pointer alias analysis,” *ACM Trans. Program. Lang. Syst.*, vol. 21, pp. 848–894, July 1999.

[148] 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.

[149] D. Grove and C. Chambers, “A framework for call graph construction algorithms,” *ACM Trans. Program. Lang. Syst.*, vol. 23, no. 6, pp. 685–746, 2001.

[150] 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.

[151] “Existing studies on the benefits of pointer analysis.” `http://lists.llvm.org/pipermail/llvm-dev/2016-March/096851.html`, 2016.

[152] B. G. Ryder, “Incremental data flow analysis,” in *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL ’83, (New York, NY, USA), pp. 167–176, ACM, 1983.

[153] R. Smith, A. Sloman, and J. Gibson, “Poplog’s two-level virtual machine support for interactive languages,” *Research Directions in Cognitive Science Volume 5: Artificial Intelligence.*, pp. 203–231, 1992.

[154] “WALA.” `http://wala.sourceforge.net/wiki/index.php/Main_Page`, 2018.

[155] 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.

[156] “IPA.” `https://github.com/april1989/Incremental_Points_to_Analysis`, 2018.

[157] 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.

[158] “SSA-based IR in WALA.” `https://github.com/wala/WALA/wiki/Intermediate-Representation-(IR)`, 2018.

[159] V. C. Sreedhar, M. Burke, and J.-D. Choi, “A framework for interprocedural optimization in the presence of dynamic class loading,” in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI ’00, (New York, NY, USA), pp. 196–207, ACM, 2000.

[160] B. Livshits, J. Whaley, and M. S. Lam, “Reflection analysis for java,” in *Proceedings of the Third Asian Conference on Programming Languages and Systems*, pp. 139–160, 2005.

[161] Y. Li, T. Tan, Y. Sui, and J. Xue, “Self-inferencing reflection resolution for java,” in *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586*, (New York, NY, USA), pp. 27–53, Springer-Verlag New York, Inc., 2014.

[162] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini, “Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders,” in *Proceedings of the*

33rd International Conference on Software Engineering, ICSE '11, (New York, NY, USA), p. 241–250, Association for Computing Machinery, 2011.

[163] "Pointer Analysis in WALA." `http://wala.sourceforge.net/wiki/\index.php/UserGuide:PointerAnalysis`, 2017.

[164] 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.

[165] Y. H. P.C. Chen, "SANDROS: a dynamic graph search algorithm for motion planning," *IEEE Transactions on Robotics and Automation*, vol. 4, pp. 390–403, 1998.

[166] N. Malone, K. Lesser, M. Oishi, and L. Tapia, "Stochastic reachability based motion planning for multiple moving obstacle avoidance," in *Proceedings of the 17th International Conference on Hybrid Systems: Computation and Control*, pp. 51–60, 2014.

[167] "Language Server Protocol." `https://langserver.org/`, 2018.

[168] "WALA projects meant for language-independent IDE support." `https://github.com/wala/IDE`, 2018.

[169] M. Sagiv, T. Reps, and R. Wilhelm, "Solving shape-analysis problems in languages with destructive updating," *ACM Trans. Program. Lang. Syst.*, vol. 20, p. 1–50, jan 1998.

[170] "DAO Hack." `https://www.insider.com/dao-hacked-ethereum-crashing-in-value-tens-of-millions-allegedly-stolen-2016-6`, 2016.

[171] "Nasdaq's Facebook glitch came from 'race conditions'." `https://www.computerworld.com/article/2504676/nasdaq-s-facebook-glitch-came-from--race-conditions-.html`, 2022.

[172] "Therac-25." `https://en.wikipedia.org/wiki/Therac-25`, 2022.

[173] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, pp. 558–565, July 1978.

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

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

[176] C. Demetrescu, *Fully Dynamic Algorithms for Path Problems on Directed Graphs*. PhD thesis, University of Rome, 2001.

[177] G. F. Italiano, Y. Nussbaum, P. Sankowski, and C. Wulff-Nilsen, "Improved algorithms for min cut and max flow in undirected planar graphs," STOC '11, (New York, NY, USA), p. 313–322, Association for Computing Machinery, 2011.

[178] "The Language Server protocol." `https://langserver.org/`, 2018.

[179] S. Park, S. Lu, and Y. Zhou, "Ctrigger: Exposing atomicity violation bugs from their hiding places," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, (New York, NY, USA), p. 25–36, Association for Computing Machinery, 2009.

[180] G. Li, S. Lu, M. Musuvathi, S. Nath, and R. Padhye, "Efficient scalable thread-safety-violation detection: Finding thousands of concurrency bugs during testing," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, (New York, NY, USA), p. 162–180, Association for Computing Machinery, 2019.

[181] Y. Yu, T. Rodeheffer, and W. Chen, "Racetrack: Efficient detection of data race conditions via adaptive tracking," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, (New York, NY, USA), p. 221–234, Association for Computing Machinery, 2005.

[182] J. Huang, P. O. Meredith, and G. Rosu, "Maximal sound predictive race detection with control flow abstraction.," in *PLDI* (M. F. P. O'Boyle and K. Pingali, eds.), pp. 337–348, ACM, 2014.

[183] A. Loginov, V. Sarkar, J. deok Choi, J. deok Choi, A. Logthor, and I. V. Sarkar, "Static datarace analysis for multithreaded object-oriented programs," tech. rep., IBM Research Division, Thomas J. Watson Research Centre, 2001.

[184] P. Pratikakis, J. S. Foster, and M. Hicks, "Locksmith: Context-sensitive correlation analysis for race detection," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, (New York, NY, USA), p. 320–331, Association for Computing Machinery, 2006.

[185] "LLVM." `http://llvm.sourceforge.net/wiki/index.php/Main_Page`, 2018.

[186] K. T. Tekle and Y. A. Liu, "Precise complexity guarantees for pointer analysis via datalog with extensions," *CoRR*, vol. abs/1608.01594, 2016.

[187] M. Sagiv, T. Reps, and R. Wilhelm, "Solving shape-analysis problems in languages with destructive updating," *ACM Trans. Program. Lang. Syst.*, vol. 20, pp. 1–50, Jan. 1998.

[188] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff, "Escape analysis for java," in *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, 1999.

[189] R. O'Callahan and J.-D. Choi, "Hybrid dynamic data race detection," in *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, (New York, NY, USA), p. 167–178, Association for Computing Machinery, 2003.

[190] W. Song, X. Qian, and J. Huang, "EHBDroid: Beyond GUI Testing for Android Applications," in *The 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 27–37, 2017.

[191] "Processes and threads overview | Android Developers." `https://developer.android.com/guide/components/processes-and-threads`, 2020.

[192] "Coderrect race detector." `https://coderrect.com/download/`, 2021.

[193] E. Pozniansky and A. Schuster, "Efficient on-the-fly data race detection in multithreaded c++ programs," in *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, (New York, NY, USA), p. 179–190, Association for Computing Machinery, 2003.

[194] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan, "Efficient and precise datarace detection for multithreaded object-oriented programs," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, (New York, NY, USA), p. 258–269, Association for Computing Machinery, 2002.

[195] D. Devecsery, P. M. Chen, J. Flinn, and S. Narayanasamy, "Optimistic hybrid analysis: Accelerating dynamic analysis through predicated static analysis," *SIGPLAN Not.*, vol. 53, p. 348–362, mar 2018.

[196] R. O'Callahan and J.-D. Choi, "Hybrid dynamic data race detection," in *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, (New York, NY, USA), p. 167–178, Association for Computing Machinery, 2003.