PUS: A Fast and Highly Efficient Solver for Inclusion-based Pointer Analysis

Peiming Liu
Texas A&M University
College Station, USA
peiming@tamu.edu

Yanze Li
Texas A&M University
College Station, USA
yanzeli@tamu.edu

Brad Swain
Texas A&M University
College Station, USA
brad@tamu.edu

Jeff Huang
Texas A&M University
College Station, USA
jeff@cse.tamu.edu

ABSTRACT

A crucial performance bottleneck in most interprocedural static analyses is solving pointer analysis constraints. We present Pus, a highly efficient solver for inclusion-based pointer analysis. At the heart of Pus is a new constraint solving algorithm that significantly advances the state-of-the-art. Unlike the existing algorithms (i.e., wave and deep propagation) which construct a holistic constraint graph, at each stage Pus only considers *partial* constraints that causally affect the final fixed-point computation. In each iteration Pus extracts a small *causality subgraph* and it guarantees that only processing the causality subgraph is sufficient to reach the same global fixed point. Our extensive evaluation of Pus on a wide range of real-world large complex programs yields highly promising results. Pus is able to analyze millions of lines of code such as PostgreSQL in 10 minutes on a commodity laptop. On average, Pus is more than 7× faster in solving context-sensitive constraints, and more than 2× faster in solving context-insensitive constraints compared to the state of the art wave and deep propagation algorithms. Moreover, Pus has been used to find tens of previous unknown bugs in high-profile codebases including Linux, Redis, and Memcached.

KEYWORDS

Static Analysis, Pointer Analysis, Causality Subgraph

ACM Reference Format:
Peiming Liu, Yanze Li, Brad Swain, and Jeff Huang. 2022. PUS: A Fast and Highly Efficient Solver for Inclusion-based Pointer Analysis. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510075

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510075

1 INTRODUCTION

Pointer alias analysis is a fundamental technique in an enormous amount of program analysis applications, such as value-flow analyses [4, 26, 32], deep bug detectors [13, 16, 18, 20], memory leak detectors [6, 8, 33, 35], etc. It is also the prerequisite of many compiler optimizations such as loop optimization and dead code elimination.

Although pointer analysis has been a focus of research for decades, it remains an open challenge to scale pointer analysis to large complex codebases. A crucial performance bottleneck is in solving the pointer analysis constraints. While precise pointer analysis is known to be undecidable [12, 24], any practical solution must over-approximate the exact answer. A state-of-the-art approach is the Andersen-style [1], *inclusion-based pointer analysis*, in which pointer assignments are constrained by inclusive relations. For example, a simple assignment $q = p$ from pointer p to q produces the contraint $pts(p) \subseteq pts(q)$, meaning that the points-to set of p, denoted as $pts(p)$, is a subset of points-to set of q. For a complex assignment involving pointer dereference, $q = *p$, it produces $\forall v \in pts(p) : pts(v) \subseteq pts(q)$. These inclusive constraints, while ensuring valid may-alias results, provide significantly higher precision than unification-based approaches (*e.g.,*Steensgaard-style [31]).

As real-world programs often produce a huge number of constraints, quadratic to the number of pointers, the key challenge remained is how to efficiently solve these constraints. There was a significant effort over a decade ago by Pereira, Hardekopf, Pearce [10, 22, 23]. In their work, a naïve fixed-point algorithm is improved by separating complex constraints and propagating the points-to information into two stages; by applying different strongly connected component (SCC) detection strategies, *e.g.,* lazy cycle detection and hybrid cycle detection, to reduce the size of the constraint graph [10]; or by sorting the constraint graph topologically to avoid redundant computation [10, 22]. More recently, Lei et al. [15] propose an efficient algorithm (Dea) for handling positive weight cycles in field-sensitive pointer analysis. Liu et al. [18] propose an incremental pointer analysis (D4) that only analyzes the updated code changes to dodge the performance overhead introduced by a whole-program pointer analysis. While these approaches further improve the state-of-the-art in some specific aspects, their fundamental solving algorithm remains the same (*e.g.,* Dea still relies on Wp [23] to solve the constraints).

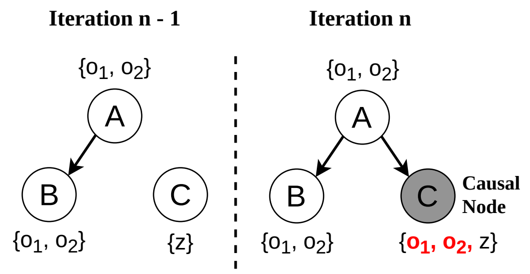


Figure 1: An example to illustrate the causality subgraph: with new edge inserted after iteration $n - 1$, node C is identified as a causal node in iteration n.

In this paper, we tackle this tremendous challenge with a new fundamental solving algorithm. Unlike previous algorithms, our new algorithm, *Partial Update Solver* (Pus), only processes a *partial* constraint graph in each iteration, yet still guarantees the same *global* fixed point. The key insight behind our approach is that during the constraint solving process in each iteration, only a very small *causality subgraph* is subject to change due to the updates made in previous iterations. With the causality subgraph, Pus prunes the constraint graph to only operate on a small subset of the constraints in each iteration, which eliminates redundant computation across iterations, resulting in a much faster algorithm.

Compared to prior approaches [10, 20, 23] that apply general graph processing techniques to pointer analysis, Pus is more efficient because it leverages two unique properties of pointer analysis:

- First, the sparsity of the constraint graph, which leads to our definition of causality subgraph;
- Second, the interconnections between different solving iterations provide the necessary information to minimize the set of causal constraints in the next iteration.

As illustrated in Fig. 1, suppose a new edge $A \longrightarrow C$ is inserted in the previous iteration (due to complex constraints), C is identified as a *causal* node because the points-to information carried by A will *take effect* on C in the current iteration in order to satisfy the inclusive constraints. However, B is not a *causal* node because its points-to information is not affected by the new edge. Our empirical results show that, on average, the causality subgraph includes less than *4%* of the nodes and edges in the full constraint graph, indicating a dramatic performance optimization opportunity (the formal definition of causality subgraph is given by **Definition 4.3**).

Fig. 2 shows an overview of Pus. At a high level, Pus adopts a similar workflow to the existing *two-phase* constraint solving algorithms [23], in which the constraints are processed iteratively between two stages (for processing simple constraints and complex constraints, respectively). However, unlike the existing algorithms, which repeat the computation over the whole graph in each iteration, Pus interactively invokes the two processing phases such that the first phase computes a causality subgraph and selectively propagates the points-to information within the causality subgraph (based on the information provided by the second phase). Meanwhile, as new points-to information is propagated, the first phase also collects a subset of all the complex constraints to be processed in the second phase. In each iteration of Pus, one phase provides

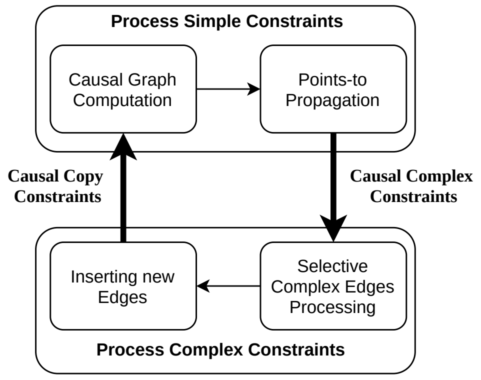

Process Simple Constraints
Causal Graph Computation
Points-to Propagation
Causal Copy Constraints
Causal Complex Constraints
Inserting new Edges
Selective Complex Edges Processing
Process Complex Constraints

Figure 2: An overview of Pus: partial update solver.

necessary information for the other to infer a small set of causal constraints to be processed.

In principle, the time complexity of Andersen-stype pointer analysis is bounded by $O(N^2 max_x D(x) + NE)$ on a *k-sparse* program [30], where $max_x D(x)$ is the maximal number of statements dereferencing a pointer x, N/E is the number of nodes/edges in the constraint graph. The value of $max_x D(x)$ is bounded by a constant k (*i.e.,* $max_x D(x) \le k$) for real-world applications. The first portion, $O(N^2 max_x D(x))$, summarizes the complexity for handling complex constraints and the second portion, $O(NE)$, summarizes the complexity for propagating points-to information on the constraint graph. As Pus propagates points-to information only on the causality subgraph, it reduces the second portion to $O(N^2 max_x D(x) + N^* E^*)$, where N^*/E^* is the number of nodes/edges in the causality subgraph. In practice, this reduction leads to significant performance improvements because typically $N^* \ll N$ and $E^* \ll E$ in real-world programs.

In summary, this paper makes the following contributions:

- We propose Pus, a novel constraint solving algorithm for inclusion-based pointer analysis. Pus identifies a minimal causality subgraph to maximize performance while ensuring that the same global fixed point is reached.
- We have proved the correctness of Pus. Formal proofs are provided in the supplementary materials.
- We conduct extensive experiments and show that Pus is more than 7× faster than the state-of-the-art Wp (Wave Propagation) and Dp (Deep Propagation) algorithms [23] in solving context-sensitive pointer analysis, and more than 2× faster in solving context-insensitive pointer analysis.
- Pus has enabled a commercial static analyzer and used in [19] to find tens of previous unknown bugs in large complex systems including Linux, Redis and Memcached (see https://coderrect.com/openscan/).

2 BACKGROUND

In this section, we introduce the background of inclusion-based pointer analysis.

Inclusion-based Pointer Analysis. The inter-procedural inclusion-based pointer analysis abstracts different program statements into the constraints listed in Table 1. It first scans the target program

Table 1: Constraints for inclusion-based pointer analysis

Category	Type	Statement	Constraints
Base	Address Taken	$v_1 \leftarrow \&o$	$loc(o)^1 \in pts(v_1)$
Simple	Assignment	$v_1 \leftarrow v_2$	$pts(v_1) = pts(v_2)$
Simple	Phi Assignment	$v \leftarrow \phi(v_1,\ v_2,\ ...)$	$pts(v) = (pts(v_1) \cup pts(v_2) \cup ...)$
Simple	Call Assignment	$r \leftarrow f(v_1, v_2, ...)$	$\forall$ **return** v *in* $f(x_1, x_2, ...) : pts(r) = pts(v) \wedge pts(x_1) = pts(v_1) \wedge pts(x_2) = pts(v_2) \wedge ...$
Complex	Load	$v_1 \leftarrow *v_2$	$\forall v \in pts(v_2) : pts(v_1) = pts(v)$
Complex	Store	$*v_1 \leftarrow v_2$	$\forall v \in pts(v_1) : pts(v) = pts(v_2)$
Complex	Offset	$v \leftarrow \&s.field$	$\forall v \in pts(s) : loc(v.field) \in pts(v)$

[1] $loc(o)$ denotes the memory location of object o.

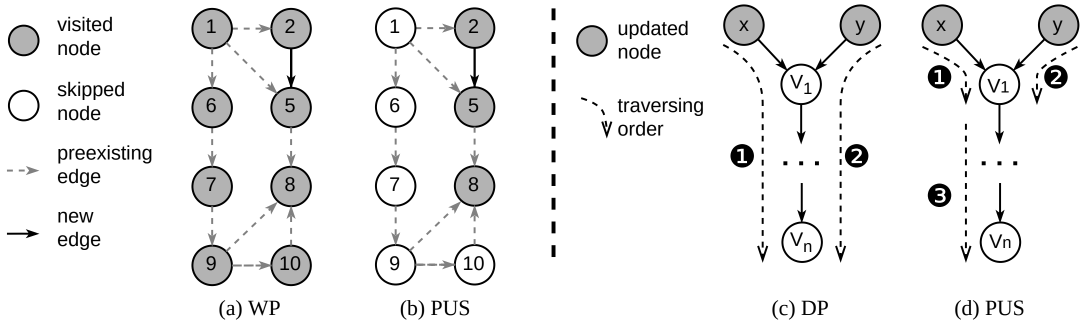


Figure 3: The comparison between Pus, Wp and Dp. (a) the solving process of Wp (the entire graph need to be revisited) (b) the solving process of Pus (only marked node need to be visited) (c) the solving process of Dp ($V_1 ... V_n$ are visited twice) (d) the solving process of Pus ($V_1 ... V_n$ are only visited once).

and generates three types of constraints: *base*, *simple* and *complex* [10]. It then abstracts the target program into a constraint graph (Definition. 2.1). Inclusion-based pointer analysis can then be solved by computing the transitive closure of the constraint graph such that for every pair of nodes $v_1, v_2 \in \mathcal{V}$, if there is an edge $e = \{v_1 \rightarrow v_2\} \in \mathcal{E}$, then $pts(v_1)$ and $pts(v_2)$ are the minimal points-to sets that ensure $pts(v_1) \subseteq pts(v_2)$.

The global fixed point is reached when all complex constraints and simple constraints are satisfied (complex constraints are satisfied by inserting new edges into the constraint graph): For each load constraint ($v_1 \leftarrow *v_2$) and every $v \in pts(v_2)$, we added a new edge $v \rightarrow v_1$ into the constraint graph; for each store constraint ($*v_1 \leftarrow v_2$) and every $v \in pts(v_1)$, we added a new edge $v_2 \rightarrow v$ into the constraint graph; and for each offset constraint ($v \leftarrow \&s.field$) and every $v \in pts(s)$, we insert $loc(v.field)$ into $pts(v)$.

Definition. 2.1: The ***constraint graph*** *(CG) of a program is an attributed graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}, pts\}$, in which $\mathcal{V}$ is a set of vertices, each of which corresponds to a variable v in the program; $\mathcal{E} \subseteq (\mathcal{V} \times \mathcal{V})$ is a set of directed edges (constraints) between vertices in $\mathcal{V}$, each of which represents a simple constraint between two nodes (in the following text, the word 'edge' and 'constraint' are used interchangeably); and pts: $\mathcal{V} \rightarrow P(O)$ (where $P(O)$ is the power set of the set of objects created by memory allocation operations in the program) is a function from $v \in \mathcal{V}$ to $s \in P(O)$ that maps a node (pointer) to its points-to set.*

3 LIMITATION OF THE EXISTING METHODS

We divide the existing constraint solving algorithms for inclusion-based pointer analysis roughly into two categories and summarize their limitations as follows respectively.

Methods that process constraints in topological order: Performing a topological sorting on the constraint graph ensures that constraints are processed in the optimal order by guaranteeing that the points-to sets of all the predecessors of a node n have been updated before processing n. In this way, the points-to sets of the predecessors are the most recently updated before propagating to the node n. Many algorithms [7, 11, 21, 22] adopt the topological sorting approach to boost the constraint solving time. Despite of the benefits, performing SCC detection and topological sorting on large constraint graphs itself is time-consuming and could easily become a bottleneck that slows down the solving process.

Fig. 3 (a) and Fig. 3 (b) shows the solving process of Wp and Pus on the example constraint graph respectively, as a new edge ($2 \rightarrow 5$) is inserted, Wp revisits the entire graph again in topological order, on the other hand, Pus computes the same result by only visiting the three nodes in the causal subgraph (marked in grey).

Methods that process constraints in undetermined order: Methods that do not enforce SCC detection and topological order on the constraint graph (*e.g.*, Deep Propagation (DP) [23], Lazy Cycle Detection (LCD) and Hybrid Cycle Detection (HCD) [10]) at each iteration unavoidably waste resources on redundant computation due to a suboptimal order of constraint processing.

Fig. 3 (c) and Fig. 3 (d) show the solving process of Dp and Pus on the example constraint graph respectively. When both $pts(x)$ and $pts(y)$ are updated, Dp adopts a depth-first search to propagate from $x \rightarrow \cdots \rightarrow v_n$ and from $y \rightarrow \cdots \rightarrow v_n$ separately. As a result, the nodes and constraints between $v_1 \rightarrow \cdots \rightarrow v_n$ are visited twice. However, Pus shows that when analyzing the graph in topological order (*i.e.,* $x \rightarrow v_1, y \rightarrow v_1$ and then $v_1 \rightarrow \cdots \rightarrow v_n$), every constraint only needs to be visited once.

The comparison between the existing two categories of algorithms reveals the dilemma of current algorithms: On one hand, full SCC detection and topological sorting are desired to eliminate redundant computation and to reduce the number of nodes by collapsing nodes in the same SCC in the constraint graph; on the other hand, applying a complete SCC detection on a large graph itself can introduce an unbearable overhead.

We found that the common problem for those works is that they all take a holistic view towards constraint graphs. Instead, Pus works on *causality subgraphs*. By only working on a small subgraph in each iteration, Pus can enjoy the benefit brought by topological sorting without introducing too much performance overhead. The rationale behind causality subgraphs and the unique interconnection between different phases of the solving process are summarized as follows:

- Constraint graphs for real programs are, by nature, *sparsely connected*. The sparsity of constraint graphs is a result of *modularization* of modern software (thus fewer connections between different modules) as well as the *locality* [1] of program statements (thus fewer connections between different statements). As constraint graphs are abstracted from programs, an update on one specific node in the constraint graph will likely only affect a limited number of neighboring nodes. Thus, in each iteration during the solving process and with limited nodes whose points-to sets are updated, only a very small subset (usually $\leq 4\%$ according to our experiments) of the nodes (casual nodes) are required to be processed, which means that topological sorting and points-to set propagation only need to be done on a small *causality subgraph* of the entire constraint graph in each iteration.

Being able to precisely infer a small subgraph in each iteration, Pus discovers another memory optimization opportunity: One of the most widely adopted optimization techniques used in existing methods and frameworks (*e.g.,* WALA [34]) is to maintain a cached points-to set for every node in the constraint graph (Wave Propagation [23] even requires an additional cached points-to set for every edge in the graph). The cached points-to set is used to filter out *non-causal nodes* whose points-to sets do not get updated in the current iteration and to only process diffed points-to information. However, if the *causality subgraph* can be accurately inferred and most of the constraints in the subgraph are effective, *i.e.,* by processing which, the points-to set will get updated, then the cached points-to set can be optionally eliminated to improve the memory efficiency without causing significant performance overhead.

[1]The locality here has different meaning from the spatial/temporal locality in computer architecture. Here, it is used to explain that most of the statements in the program are irrelevant (*e.g.,* `a++;` and `b++;`).

4 ALGORITHM

In this section, we describe the detailed algorithm for Pus. We first present the overall structure of Pus in Algorithm 1, we then explain each component separately in detail in Algorithm 2, Algorithm 3 and Algorithm 4. For simplicity, we describe Pus under the context of *field-insensitive* pointer analysis. Pus can be extended for field-sensitive pointer analysis (as we implemented for experiments) by adding another type of constraint, i.e., the *offset* constraint, into complex constraints similar to the previous work [22].

4.1 Structure of the Algorithm

At a high level, Pus has a similar structure to Wp [23] that separates the insertion of new constraints (handling complex constraints) from the propagation of points-to sets (handling simple constraints). However, Pus distinguishes itself by connecting the two constraint solving phases using two separate work lists:

- L_{copy}:$\{\mathcal{E}\}$ – A subset of simple constraints that is used to compute the causality subgraph used in following stages.

Algorithm 1: Partial Update Solver

```
Input  : A unsolved constraint graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}, pts\}$
Result : The points-to information for every pointer in the
         program
1  $L_{comp} \leftarrow \varnothing$;
2  $L_{copy} \leftarrow \varnothing$;
3  for each $v \in \mathcal{V}$ do
4      if $pts(v) \neq \varnothing$ then
           // Nodes with address taken constraints
              have non-empty points-to set
5          $C_{copy} \leftarrow v$.getCopyConstraints();
           // get simple constraints started from the
              node and insert them into $L_{copy}$
6          $L_{copy}$.insert($C_{copy}$);
           // insert the node into $L_{comp}$
7          $L_{comp}$.insert($v$);
8      end
9  end
10 while $L_{copy} \neq \varnothing$ do
       // SCC collapse on subgraphs of $\mathcal{G}$ based on
          $L_{copy}$
11     SCC Collapse and TopoSort on subgraphs of $\mathcal{G}$
        ($Algorithm\ 2$);
12     $L_{comp} \leftarrow$ Partially Process Simple Constraints
        ($Algorithm\ 3$);
13     $L_{copy}.clear()$;
14     $L_{copy} \leftarrow$ Partially Process Complex Constraints
        ($Algorithm\ 4$);
15     $L_{comp}.clear()$;
16 end
```

- L_{comp}:$\{\mathcal{V}\}$ – A subset of nodes on which the complex constraints need to be recomputed.

At a high level, Algorithm 1 can be divided into initialization phase (from line 3 to 9), SCC detection and topological sort phase (line 11), Simple constraint processing phase (line 1) and Complex constraint processing phase (line 14), which are explained in detail in the following sections. We also relies on the following conventions to describe our algorithm: We refer to any edge $e \in L_{copy}$ used in Algorithm 1 as an **essential edge** and refer to any node $v \in L_{comp}$ used in Algorithm 1 as an **unsaturated node**. We use $dst(\mathcal{E})$ to denote the set of destination nodes for all edges $e \in \mathcal{E}$; we use $src(\mathcal{E})$ to denote the set of source nodes for all $e \in \mathcal{E}$; we use $in(n)$ to denote the set of incoming edges to node n; we use $out(n)$ to denote the set of outgoing edge from n; we use $pred(n)$, where n is a node, to denote the set of predecessor nodes of n; we use $succ(n)$ to denote the set of the successor nodes of n.

4.2 Detailed Algorithm

In this section, we describe the detailed algorithms of all subcomponents that are used in Algorithm 1.

Algorithm 2: SCC Collapse and TopoSort on SubGraphs of G

```
Input  : A constraint graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}, pts\}$
         A list of starting edges $L_{copy} = \{\mathcal{E}\}$
Output : A toposorted vector $\mathcal{V}$ of SCCs that are reachable
         from at least one of $e \in L_{copy}$
1  $\mathcal{V}' \leftarrow \varnothing$;
2  $\mathcal{E}' \leftarrow \varnothing$;
3  $pts' \leftarrow pts$;
4  $\mathcal{G}' \leftarrow \{\mathcal{V}', \mathcal{E}', pts'\}$;
   // prune the graph $\mathcal{G}$ to a subgraph $\mathcal{G}'$
5  while $L_{copy} \neq \varnothing$ do
6      $e \leftarrow L_{copy}$.pop();
7      if visited(e) then
8          continue ; // skip covered edges
9      end
10     setVisited($e$);
11     $\mathcal{E}'$.insert($e$);
       // add source and destination nodes of $e$ into
          $\mathcal{G}'$
12     $\mathcal{V}'$.insert($\{e.src, e.dst\}$);
13     $\mathcal{E}'$.insert($\{e' \mid e' \in$
         $v.outgoing\_edges() \wedge reachable(e.dst, v) = true\}$);
14     $\mathcal{V}'$.insert($\{v' \mid reachable(e.dst, v) = true\}$);
15 end
   // perform SCC detection on the subgraph $\mathcal{G}'$
   // also sort the graph internally
16 $\mathbb{V} \leftarrow$ Tarjan($\mathcal{G}'$);
   // return the toposorted vector $\mathbb{V}$
17 return $\mathbb{V}$;
```

Subgraph SCC Detection: As shown in Algorithm 2, the SCC detection is performed on the subgraph $\mathcal{G}'$ instead of the original graph $\mathcal{G}$. The set of edges and nodes in $\mathcal{G}'$ is computed according to the reachability from the constraints in L_{copy}.

The node set $\mathcal{N}'$ of $\mathcal{G}'$ consists of (1) the *source* and *destination* nodes of every constraints in L_{copy} and (2) all the nodes that are reachable for at least one of the *destination* nodes of the constraints in L_{copy} . The edge set $\mathcal{E}'$ of $\mathcal{G}'$ consists of (1) all the edges in L_{copy} and (2) all the *outgoing* edge of node n that are reachable from a least one of the *destination* nodes of the constraints in L_{copy}.

After SCC detection, a vector of nodes in topological order is returned by Algorithm 2 and used as one of the inputs for Algorithm 3. Note that although Algorithm 2 presents the computation of $\mathcal{G}'$ as a separate step, $\mathcal{G}'$ can be computed along with SCC detection utilizing the DFS traversal performed by Tarjan's algorithm internally.

Propagating points-to set on the causality subgraph: Algorithm 3 describes the procedure for processing simple constraints. Algorithm 3 takes a subgraph $\mathcal{G}'$ of $\mathcal{G}$ and a topologically sorted

Algorithm 3: Partially Process Copy Constraints

```
Input  : A constraint graph: $\mathcal{G}' = \{\mathcal{V}', \mathcal{E}', pts'\}$
         A sorted vector of SCCs: $\mathbb{V} = \{\mathcal{N}\}$
         A list of effective copy constraints: $L_{copy}$
Output : A set of node $\mathcal{S}$ whose complex constraints need
         to be processed
1  $L_{comp} \leftarrow \varnothing$;
2  while $\mathbb{V}$.isNotEmpty() do
3      $n \leftarrow \mathbb{V}$.pop();
4      for each $e = \{src, dst\} \in n.getCopyConstraits()$ do
5          if $e = \{src, dst\} \in L_{copy} \vee src \in L_{comp}$ then
6              $changed \leftarrow$ PropagatePointsTo(src, dst);
7              if $changed$ then
8                  $L_{comp}$.insert(dst);
9              end
10         else
               // Prune the graph, skip unchanged
                  subgraph
11             continue;
12         end
13     end
14 end
15 return $L_{comp}$;
   /* Process a simple constraint between src and
      dst, return true if the points-to information
      is updated                                    */
16 Function PropagatePointsTo($src, dst$):
17     $pts(dst) \leftarrow pts(dst) \cup pts(src)$;
18     if $dst.changed$ then
19         return true;
20     end
21     return false;
22 End Function
```

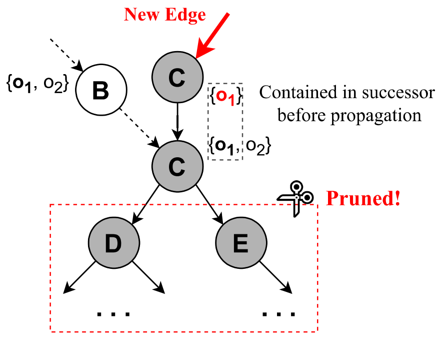


Figure 4: Further prune on the constraint graph during simple constraints processing phase.

vector of nodes as the inputs. The topologically sorted vector of nodes ensures that simple constraints are processed in the optimal order to avoid redundant computation. L_{copy} is also passed in and used at line 5 to perform further pruning on the causality subgraph.

There are two important details that are worth noting in Algorithm 3:

(1) During the points-to set propagation, the algorithm also computes and eventually outputs a list of nodes, L_{comp}, to be used in Algorithm 4, which contains all the nodes on which the complex constraints need to be processed.
(2) At line 11, the algorithm performs another pruning on the causality subgraph to further reduce the number of constraints processed by Pus.

The computation on L_{comp} is straightforward, Algorithm 3 simply inserts a node into L_{comp} if the points-to set of the node has been updated during the current iteration.

The graph processed by Algorithm 3 defines the causality subgraphs in each iteration. In addition, we introduced the following definition to formally define the causality subgraph.

Algorithm 4: Partially Process Complex Constraints

```
Input  : The constraint graph: $\mathcal{G} = \{\mathcal{V}, \mathcal{E}, pts\}$
         Nodes with effective complex constraints: $L_{comp}$
Output : A set of processed: $L_{copy}$
while $L_{comp}$.isNotEmpty() do
    $V \leftarrow L_{comp}$.pop();
    for each $\{l \leftarrow *V\} \in V$.getLoadConstraints() do
        // process load constraints
        $newEdges \leftarrow$ processLoad($l, V$);
        $L_{copy}$.insert($newEdges$);
    end
    for each $\{*V \leftarrow r\} \in V$.getStoreConstraints() do
        // process store constraints
        $newEdges \leftarrow$ processStore($V, l$);
        $L_{copy}$.insert($newEdges$);
    end
end
return $L_{copy}$;
```

Definition. 4.1: Given a constraint graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}, pts\}$ and a set of essential edges $\mathcal{E}^+ \subseteq \mathcal{E}$, the ***essential-edge-covered graph*** *$\mathcal{G}' = \{\mathcal{V}', \mathcal{E}', pts'\}$ is a subgraph of $\mathcal{G}$, where $\mathcal{V}' = \mathcal{V}_1 \cup src(\mathcal{E}^+)$ and $\mathcal{V}_1 = \{v \mid \exists s \in dst(\mathcal{E}^+), v \text{ is reachable from } s\}$; $\mathcal{E}' = \mathcal{E}^+ \cup \{e \mid e = out(n) \ \wedge \ n \in \mathcal{V}_1\}$ and $pts' = pts$.*

Definition 4.2: Given an ***essential-edge-covered graph*** *$\mathcal{G}' = \{\mathcal{V}', \mathcal{E}', pts'\}$ and its corresponding essential edge set $\mathcal{E}^+ \subseteq \mathcal{E}'$, the set of* ***ineffective edges*** *$\mathcal{E}^-$ and the set of* ***ineffective nodes*** *$\mathcal{V}^-$ are determined dynamically during the points-to set propagation process. For node n, if $\forall p \in pred(n)$, $pts(p)$ does not get updated in the current iteration, then $n \in \mathcal{V}^-$. Similarly, $\mathcal{E}^- = \{e \mid e \in out(n) \wedge n \in \mathcal{V}^- \wedge e \notin \mathcal{E}^+\}$.*

Definition 4.3: Given an ***essential-edge-covered graph*** *$\mathcal{G}' = \{\mathcal{V}', \mathcal{E}', pts'\}$ and a set of ineffective edges $\mathcal{E}^-$, the* ***causality subgraph*** *$\mathcal{G}^* = \{\mathcal{V}^*, \mathcal{E}^*, pts^*\}$,which is processed by Pus, is a subgraph of $\mathcal{G}'$, where $\mathcal{V}^* = \mathcal{V}' - \mathcal{V}^-$, $\mathcal{E}^* = \mathcal{E}' - \mathcal{E}^-$, and $pts^* = pts'$.*

Intuitively, definition 4.2 defines the set of nodes and edges that are pruned in Algorithm 3 at line 11, and the causality subgraph is defined by excluding the pruned nodes and edges from the *essential-edge-covered graph*. Fig. 4 offers an example that explains the rationale behind the graph pruning. In Fig. 4, the grey nodes and solid edges are within the essential-edge-covered graph $\mathcal{G}'$ for the current iteration. The corresponding points-to set is marked beside each node. In this example, the incoming update ($\{O_1\}$) to be propagated within the causality subgraph is already included in $pts(C)$ due to $B \longrightarrow C$. To further propagate the points-to set from C does not make any update to C's successors (D and E in the example), thus the causality subgraph can be pruned by skipping $C \longrightarrow D$ and $C \longrightarrow E$. In Algorithm 3, since nodes are processed in topological order and all the nodes whose points-to sets have been updated in the current iteration are in L_{comp}, the test on $src \in L_{comp}$ at line 5 returns true only when $pts(src)$ gets updated in the current iteration. For node dst, if all the predecessors of dst are not included in L_{comp} and thus have not been updated, the outgoing edges of dst will be pruned.

By the end of the computation, Algorithm 3 outputs L_{comp} after draining the inputted node vector and passes L_{comp} to Algorithm 4.

Processing complex constraints: Algorithm 4 provides detailed information on how Pus handles complex constraints. The algorithm takes L_{comp}, the list of nodes provided by Algorithm 3, and locates all the nodes on which the complex constraints need to be processed.

The processing of the complex constraints follows a standard procedure as described in Section 2 by inserting new edges into the constraint graph. Algorithm 4 inserts all the newly added edges into the L_{copy} and eventually passes L_{copy} to both Algorithm 2 and Algorithm 3.

Note that whether or not a cached points-to set should be maintained so that Pus is able to process only the diffed points-to set [23] for complex constraints can be optionally applied. We omit the cached points-to set in our algorithm description as well as our implementation for better memory efficiency and our experimental results show that Pus is still much faster than techniques which apply the cached points-to set optimization.

4.3 Proof of Correctness

We prove that Pus will reach the global fixed point by the end of the computation in this section.

Definition. 4.4: We say that a constraint graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}, pts\}$ is ***points-to saturated*** *or reaches a* ***points-to saturated state*** *iff for any pair of nodes $v_1, v_2 \in \mathcal{V}$, if there is a path from v_1 to v_2, we have the minimal sets for $pts(v_1)$ and $pts(v_2)$ and $pts(v_1) \subseteq pts(v_2)$.*

Definition. 4.5: We say that a constraint graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}, pts\}$ is ***constraint saturated*** *or reaches a* ***constraint saturated state*** *iff for any node $v \in \mathcal{V}$, if there is a load constraint (`p = *v`) on v then there is an edge $e = \{v' \to p\} \in \mathcal{E}$ for every $v' \in pts(v)$; and if there are store constraints (`*v = p`) on v, then there is an edge $e = \{p \to v'\} \in \mathcal{E}$ for every $v' \in pts(v)$.*

By definition, the global fixed point is reached when the constraint graph is both points-to saturated and constraints saturated.

Lemma 4.1: Given an acyclic constraint graph, it will reach a *points-to saturated state* after processing the nodes once in topological order. □

Lemma 4.2: The *ineffective edge set $\mathcal{E}^-$* is empty during the first iteration in Algorithm 1. □

Theorem 4.1: At every iteration in Algorithm 1, the constraint graph ❶ is *points-to saturated* after processing simple constraints (line 13) and ❷ is *constraint saturated* after processing complex constraints on *unsaturated nodes* (line 14). □

Proof: We prove the theorem by induction.

For the first iteration. ❶: By **Lemma 4.2**, the first iteration processes the entire *essential-edge-covered graph $\mathcal{G}' = \{\mathcal{V}', \mathcal{E}', pts'\}$* with an essential edge set $\mathcal{E}^+ = \{e \mid e \in out(n) \wedge pts(n) \neq \emptyset\}$. By **Lemma 4.1**, the subgraph $\mathcal{G}'$ will reach a *points-to saturated state* after simple constraints processing. To prove the whole graph $\mathcal{G}$ will also be *points-to saturated*, it is equivalent to show that for nodes $n \notin \mathcal{V}'$, $pts(n) = \emptyset$: By contradiction, if there exists a node $n \notin \mathcal{V}' \wedge o \in pts(n)$, by the transitivity of constraint graph [18], there exists a path from address taken node of o to node n. However, since $\mathcal{E}^+$ includes all the address taken nodes' outgoing edges, node n should also be included in $\mathcal{V}'$ by definition, which contradicts with $n \notin \mathcal{V}'$. ❷: According to Algorithm 1, the *unsaturated node* set $\mathcal{V}^+ = \{v \mid pts(v) \neq \emptyset\}$ after processing simple constraints at line 13. It is obvious that the graph reaches a *constraint saturated state* after processing complex constraints on $\mathcal{V}^+$ as no edge needs to be inserted for node n whose points-to set is empty.

Combining ❶ and ❷, theorem 5.1 holds at the first iteration.

Suppose theorem 4.1 holds for the *n*-th iteration.

For the *n+1*-th iteration. We denote the constraint graph at n-th iteration before inserting new edges as $\mathcal{G}_n = \{\mathcal{V}_n, \mathcal{E}_n, pts_n\}$, the constraint graph at current iteration before inserting new edges as $\mathcal{G}_{n+1} = \{\mathcal{V}_{n+1}, \mathcal{E}_{n+1}, pts_{n+1}\}$ and the *causality graph* processed at current iteration as $\mathcal{G}^*_{n+1} = \{\mathcal{V}^*_{n+1}, \mathcal{E}^*_{n+1}, pts^*_{n+1}\}$

❶*:* According to Algorithm 1 and Algorithm 4, the essential edge set $\mathcal{E}^+_{n+1} = \mathcal{E}_{n+1} - \mathcal{E}_n$. To prove that a *points-to saturated state* will be reached, we prove the following two conditions hold:

(1) for node $v \in \mathcal{V}_{n+1} - \mathcal{V}^*_{n+1}$, $pts_n(v) = pts_{n+1}(v)$ and thus need not to be processed, and
(2) the pruning on $\mathcal{G}^*_{n+1}$ by removing ineffective constraints in $\mathcal{E}^-_{n+1}$ is sound.

For (1), assume there exists a node $v \in \mathcal{V}_{n+1} - \mathcal{V}^*_{n+1}$ and $\Delta_{n+1} = pts_{n+1}(v) - pts_n(v) \neq \emptyset$. By the transitivity of constraint graph, for $o \in \Delta_{n+1}$, there exists a path from the address taken node o' to v (denoted as a set $\mathcal{P} = \{o' \to v_1, v_1 \to v_2, ..., v_x \to v_y, v_y \to v\}$).

Case 1: If for every $e \in \mathcal{P}$, $e \notin \mathcal{E}^+_{n+1}$, then $e \in \mathcal{E}_n$. By induction hypothesis, the n-th iteration reached the *points-to saturated state*, thus $o \in pts_n(v)$ since there is a path $\mathcal{P}$ between o' and v, which is contradictory to the assumption $o \in \Delta_{n+1}$.

Case 2: If there exists a $e \in \mathcal{P}$, and $e \in \mathcal{E}^+_{n+1}$, then by definition $v \in \mathcal{V}^*_{n+1}$ and v is in the causality graph, which is contradictory to the assumption $v \in \mathcal{V}_{n+1} - \mathcal{V}^*_{n+1}$.

For (2), assume there exists an edge $e \in \mathcal{E}^-_{n+1}$ and by processing it, which is to compute $pts_{n+1}(e.dst) = pts_{n+1}(e.src) \cup pts_n(e.dst)$, $\Delta_{n+1} = pts_{n+1}(e.dst) - pts_n(e.dst) \neq \emptyset$. Since, by definition, $pts_{n+1}(e.src) - pts_n(e.src) = \emptyset$ as e is an ineffective edge. To satisfy $\Delta_{n+1} \neq \emptyset$, we have $pts_n(e.src) \not\subseteq pts_n(e.dst)$. However, since $e \in \mathcal{E}^-_{n+1}$, by definition $e \notin \mathcal{E}^+_{n+1}$, which equals $\mathcal{E}_{n+1} - \mathcal{E}_n$. We can conclude that $e \in \mathcal{E}_n$. By induction hypothesis, we have $pts_n(e.src) \subseteq pts_n(e.dst)$ and $e.dst$ is reachable from $e.src$ by $e \in \mathcal{E}_n$ and $\mathcal{G}$ is points-to saturated, which is contradictory to the assumption $pts_n(e.src) \not\subseteq pts_n(e.dst)$.

❷*:* According to Algorithm 3 and Algorithm 1, the set of *unsaturated nodes* $\mathcal{V}^+_{n+1} = \{v \mid pts_{n+1}(v) - pts_n(v) \neq \emptyset\}$. It is obvious that the algorithm will reach *constraint saturated state* after processing $v \in \mathcal{V}^+_{n+1}$. By induction hypothesis, in the previous iteration after inserting new edges, the constraint graph is *constraint saturated* and thus for $v' \in \{v \mid pts_{n+1}(v) = pts_n(v)\}$, they need not to be processed in the current iteration.

Combining ❶ and ❷, Theorem 4.1 holds at the $n + 1$-th iteration provided that Theorem 4.1 holds in n-th iteration.

Theorem 4.2: Algorithm 1 guarantees the global fix point. □

Proof: We denote the constraint graph in the final iteration before inserting new edges as $\mathcal{G}_f = \{\mathcal{V}_f, \mathcal{E}_f, pts_f\}$ and the constraint graph in the final iteration after inserting new edges as $\mathcal{G}'_f = \{\mathcal{V}_f, \mathcal{E}'_f, pts_f\}$ Since Algorithm 1 returns when the *essential edge set $\mathcal{E}^+_f = \mathcal{E}'_f - \mathcal{E}_f = \emptyset$*, no edge is inserted by processing new complex constraints. Thus $\mathcal{G}_f = \mathcal{G}'_f$. By Theorem 4.1, $\mathcal{G}_f$ is *points-to saturated* and $\mathcal{G}'_f$ is *constraint saturated* and since $\mathcal{G}_f = \mathcal{G}'_f$, the final output $\mathcal{G}'_f$ are both points-to saturated and constraint saturated.

5 EVALUATION

We have implemented a *context-sensitive, field-sensitive* pointer analysis for C/C++. The implementation is based on the LLVM [14] framework and works at LLVM IR level. In our evaluation, we compared Pus with Wave Propagation (Wp) and Deep Propagation (Dp) [23]. The two state-of-the-art algorithms are widely adopted in the most recent pointer analysis works [2, 3, 9].

In our implementation, we adopted the sparse bitvector to store points-to set information. We did not compare Pus with some other

Table 2: Benchmarks and the constraint graph metrics (#Pointer, #Object and #Assign shows the number of pointers, objects and assignment statements in the tested program respectively)

Benchmark	#LoC	#Pointer	#Object	#Assign
memcached	18.9K	15.2K	3.8K	6.0K
darknet	30.1K	91.3K	26.0K	44.1K
flatbuffers	156.1K	210.2K	83.5K	2659.5K
nfs-ganesha	251.5K	114.1K	33.5K	768.6K
curl	142.2K	70.0K	14.1K	578.5K
sqlite3	245.4K	129.3K	23.6K	1024.4K
keydb-server	259.1K	78.9K	20.0K	230.1K
vim	334.9K	267.9K	51.1K	1826.9K
cpython	564.9K	171.5K	52.2K	1770.0K
postgreSQL	1.0M	496.7K	106.3K	4677.6K

recent techniques (*e.g.,* D4 [18] and Dea [15]) in our evaluation as they are solving orthogonal issues to Pus. In fact, we believe that Pus can be incorporated with these techniques to provide a faster underlying solving algorithm. For instance, Dea used Wp in its implementation, which can be directly replaced with Pus.

The goal of our evaluation is to answer the following research questions.

- **RQ1:** How much reduction can Pus achieve by only processing the *causality subgraph* in each iteration? In other words, how large is the *causality subgraph* for real-world applications when compared to the entire constraint graph?
- **RQ2:** In terms of performance, how much faster is Pus when compared with state-of-the-art algorithms, namely Wp and Dp?

All our experiments are conducted on a commodity personal desktop embedded with an Intel i7-9750H processor with 6 cores @ 2.6GHz and 128GB RAM.

Benchmarks: We selected 10 representative open-sourced real-world large projects as the benchmarks to evaluate Pus. Many of them have also been studied for the similar purpose in previous publications [23]. And they are all popular open-source projects varying in size. Metrics of those benchmarks and their constraint graphs are listed in Table 2. The selected benchmarks are medium- to large-sized projects with sizes ranging from 18.9K to 1.0M lines of code.

5.1 RQ1: Reduction Achieved by Pus

To answer the first research question, we ran *context-insensitive* Pus on the benchmarks in Table 2 and collected statistics about the size of the causality subgraph processed by Pus in each iteration. The detailed report is listed in Table 3. Table 3 compares the size of different causality subgraphs and analyzes the relative sizes of the causality subgraphs compared with the entire constraint graph.

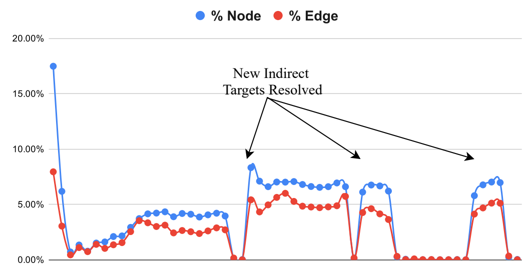
We report the *minimal*, *maximum* and *average* number of nodes and edges processed by Pus to summarize the characteristics of the causality subgraph because a different causality subgraph is computed by Pus in each iteration.

As shown in Table 3, on average a causality subgraph only contains around 3% of the nodes and 2.7% of the edges in the respective whole constraint graph. For most of the benchmarks, the size of the causality subgraph can be as small as just 1 or 2 nodes and edges, even for large benchmarks (*e.g.,* cpython and postgreSQL) with more than 500K nodes and 300K edges. The minimal causality subgraph is usually observed in the last few iterations when the points-to sets of most of the nodes in the constraint graph are saturated. The result gives us more confidence on the performance improvement can be achieved by Pus, as algorithms like Wp would still need to re-sort the entire constraint graph even when the number of effective nodes can be as low as 1.

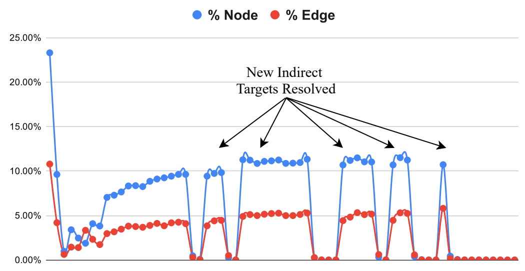
The result also shows that for most of the benchmarks, even the largest causality subgraph usually contains no more than 30% of the nodes and 30% of the edges in the complete constraint graph. More importantly, according to our observation, large causality subgraphs do not occur frequently, which is also why the average

Table 3: The size of the causality subgraphs processed by Pus (%Ratio compares the size of the causality subgraph with the whole constraint graph)

Benchmark		causality Subgraph			
		#Node	%Ratio	#Edge	%Ratio
memcached	min	1	0.01%	2	0.02%
	max	3,538	18.59%	11,175	75.53%
	avg.	**197**	**1.04%**	**703**	**4.74%**
darknet	min	1	0.00%	1	0.00%
	max	15,365	13.10%	34,406	31.10%
	avg.	**1,990**	**1.70%**	**7,171**	**6.48%**
flatbuffers	min	7	0.00%	11	0.00%
	max	48,533	16.52%	230,758	2.54%
	avg.	**6,679**	**2.27%**	**16,212**	**0.18%**
nfs-ganesha	min	1	0.00%	2	0.00%
	max	28,865	20.07%	62,491	16.61%
	avg.	**726**	**0.50%**	**2,590**	**0.69%**
curl	min	2	0.00%	2	0.00%
	max	14,743	17.48%	51,365	7.95%
	avg.	**3,341**	**3.96%**	**17,688**	**2.74%**
sqlite3	min	1	0.00%	2	0.00%
	max	34,642	23.32%	120,412	10.79%
	avg.	**9,113**	**5.97%**	**30,938**	**2.77%**
keydb-server	min	2	0.00%	2	0.00%
	max	21,147	21.36%	44,993	16.60%
	avg.	**4,865**	**4.91%**	**12,229**	**4.51%**
vim	min	2	0.00%	6	0.00%
	max	68,600	21.50%	235,240	12.42%
	avg.	**10,512**	**3.29%**	**39,570**	**2.09%**
cpython	min	16	0.00%	16	0.00%
	max	52,424	23.44%	139,810	7.53%
	avg.	**9,885**	**4.42%**	**33,676**	**1.81%**
postgreSQL	min	1	0.00%	1	0.00%
	max	114,410	18.97%	333,764	6.90%
	avg.	**13,241**	**2.20%**	**46,621**	**0.96%**
avg.		-	**3.02%**	-	**2.69%**



(a) The footprint of *curl*



(b) The footprint of *sqlite3*

Figure 5: The footprint of the size of the causality subgraphs processed by PUS at each iteration when analyzing *curl* and *sqlite3*.

number of nodes and edges in the causality graph is still low despite the existence of some relatively large subgraphs. Our experiments shows that large causality graphs normally occur in the first few iterations at the beginning of the computation and/or after indirect calls are resolved and new nodes are inserted. These observations are validated in Fig. 5 and will be elaborated in the following paragraphs.

In order to gain insights into the entire 'lifetime' of the causality subgraphs and to understand how it 'evolves' as the analysis proceeds, we include two complete (also *typical*) footprints that show how the sizes of causality subgraphs fluctuate in each iteration of the whole solving process. The two data sets are collected by evaluating PUS on *curl* and *sqlite3* and are visualized in Fig. 5 (a) and Fig. 5 (b) respectively. It is clear that Fig. 5 (a) and Fig. 5 (b) exhibit several common patterns:

- The size of the causality graph normally increases greatly as new indirect calls are resolved. This is because each time when an indirect call is resolved, the newly resolved target functions introduce many unprocessed nodes and constraints into the constraint graph. Those unprocessed nodes are likely to invalidate a large portion of the constraint graph, which in turn increases the size of the causality subgraph for the next iteration.
- After new nodes are inserted, the size of the causality subgraph normally reduces sharply after several iterations. The size then remains small until another set of new indirect calls get resolved. This indicates that the solving process converges quickly after a few iterations on most of the nodes, and then gradually approach the fixed point by only processing a very small number of nodes at each iteration.

Table 4: Performance of PUS comparing with wave propagation (WP) and deep propagation (DP) when running context-insensitive pointer analysis (%↑ shows the speedup).

Benchmark	PUS	WP		DP	
		time	%↑	time	%↑
memcached	0.04s	0.35s	775.00%	0.1s	150.45%
darknet	0.34s	1.82s	435.29%	1.00s	194.12%
flatbuffers	95.9s	195.72s	104.08%	124.87s	30.28%
nfs-ganesha	11.17s	48.83s	327.15%	26.48s	137.06%
curl	18.45s	29.45s	59.62%	28.29s	53.27%
sqlite3	46.48s	98.77s	125.50%	128.73s	176.96%
keydb-server	4.84s	7.58s	56.61%	5.76s	19.00%
vim	81.17s	183.81s	126.41%	193.70s	138.55%
cpython	400.66s	619.55s	54.61%	655.91s	63.70%
PostgreSQL	1,381.2s	1,757.9s	27.27%	2,001.6s	44.91%
avg.	-	-	**3.09×**	-	**2×**

Table 3 and Fig. 5 provide strong evidence to support our key observation: The size of causality subraphs are small and updates on the points-to information of certain nodes only affect very limited set of neighboring nodes. From these experiments, we can easily understand why PUS is able to achieve such a dramatic reduction by analyzing small causality subgraphs instead of the entire constraint graph at each iteration.

5.2 RQ2: The Performance Improvement Achieved by PUS

PUS was evaluated in both context-insensitive and context-sensitive (k-callsite, with $k = 1$) settings. and compared with WP and DP. The experimental results are elaborated in Section 5.2.1 (when running context-insensitive analysis) and Section 5.2.2 (when running context-sensitive analysis).

5.2.1 Improvement when Running Context-Insensitive Pointer Analysis. In the context-insensitive setting, the execution time of each algorithm when running on different benchmarks is given in Table 4.

In summary, PUS achieves a significant performance improvement compared to WP and DP, with more than 2× speedup on average. For certain benchmarks, namely *memcached* and *darknet*, PUS can be 4× as faster than WP. When compared with DP, PUS is 2× faster on more than half of the tested benchmarks (namely *memcached*, *darknet*, *nfs-ganesha*, *vim* and *sqlite3*). Even in the worst cases, PUS can still be more than 20% faster than DP and WP.

To understand how long it takes for PUS to finish one iteration, we also collected the analyzing time spent by PUS and WP on every iteration. The visualized graph (when analyzing *curl*) is shown in Fig. 6. From Fig. 6, it is clear that PUS is faster than WP *in every iteration* as PUS only processes a small causality subgraph. Note that DP is omitted in Fig. 6. It is because while PUS and WP use a similar *two-phases* structure and their solving processes can be easily aligned and compared. DP adopts a different solving strategy

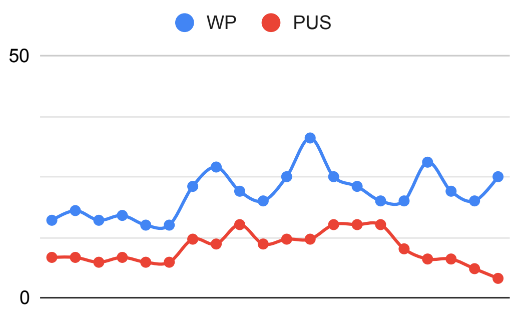


Figure 6: Comparison between the running time (in ms) of Pus and WP that is spent on every iteration when running on *curl*.

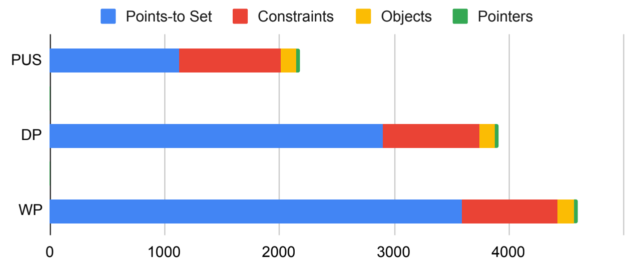


Figure 7: The memory usage breakdown for Pus, Dp and Wp on *flatbuffers* (in *MB*).

which makes the comparison between Pus and Dp meaningless when just looking at one iteration.

In our experiments, we also made a similar observation as found in the original Wp and Dp paper [23]. The original paper observes that Wp has an advantage over Dp when analyzing relatively large program as Wp is faster than Dp on *sqlite3, vim* and *cpython* and Dp is faster than Wp on relatively smaller programs such as *memcached, redis-server* and *nfs-ganesha*. The one exception is *flatbuffer*, which has a relatively small number of lines of code (48.9K) while its corresponding constraint graph is nearly as big as that of large programs such as *vim*. However, unlike Wp and Dp, which have different advantages when analyzing programs of different scales, Pus outperforms both Wp and Dp on all the tested benchmarks with the sizes ranging from $18K$ to over $1M$ lines of code. Pus can be $8\times$ faster and achieves at least a 19% speedup. The fact that Pus is able to outperform both Wp and Dp on benchmarks of varying sizes (both large and small) indicates that Pus is a much more general algorithm that can be applied to all kinds of programs.

In addition to the performance, we also evaluated the memory efficiency of Pus when compared to Dp and Wp. Fig. 7 shows typical memory usage breakdowns for Pus, Dp and Wp.

As mentioned in Section 3, Pus can greatly reduce the memory consumption because Pus does not rely on a cached points-to set to avoid redundant computation. The memory usage breakdown in Fig. 7 indicates that for both Wp and Dp, the memory used to store points-to sets (including cached points-to sets) accounts for

Table 5: Performance of Pus comparing with wave propagation (WP) and deep propagation (DP) when running k-callsite sensitive ($k = 1$) pointer analysis (%↑ shows the speedup).

Benchmark	PUS	WP		DP	
		time	%↑	time	%↑
memcached	0.08s	0.68s	708.33%	0.26s	210.71%
darknet	1.09s	6.36s	484.30%	5.33s	389.81%
flatbuffer	673.39s	4580.5s	580.22%	2542.8s	277.62%
nfs-ganesha	33.28s	367.97s	1005.82%	459.54s	1281.0%
curl	37.58s	266.98s	610.40%	258.3s	587.31%
sqlite3	112.44s	639.64s	468.88%	961.3s	754.96%
keydb-server	20.38s	77.51s	280.42%	79.66s	290.98%
vim	367.62s	2587.1s	603.75%	3647.1s	892.09%
cpython	367.33s	3358.9s	814.43%	9559.7s	2502.5%
PostgreSQL	OOM	OOM	-%	OOM	-%
avg.	-	-	**7.17×**	-	**8.99×**

the majority of the memory used to analyze a program. Since Wp requires an extra copy of the points-to set for every node *and* every edge, it requires the largest-sized memory to analyze the same program. Dp is more memory efficient when compared to Wp as it only requires an extra cached points-to set for each node, but even for Dp, the memory used to store the points-to set still accounts for the largest portion of entire used memory. Interestingly, the memory used by Dp to store the points-to set alone is larger than the entire memory consumption needed by Pus. This shows the great advantage of Pus over compared methods as it can achieve significant performance improvement while consuming much less memory.

5.2.2 *Improvement when Running Context-Sensitive Pointer Analysis.*

In the context-sensitive setting (k-callsite, with $k = 1$), the execution time of each algorithm when running on different benchmarks is given in Table 4.

Surprisingly, Pus even achieved much higher speedups when solving context-sensitive constraints when compared to context-insensitive constraints. The results shows that on average, Pus is almost $7\times$ and $9\times$ faster than Wp and Dp respectively. For certain benchmarks, Pus can be more than $10\times$ faster than Wp and Dp (*ganesha*) and more than $25\times$ faster than Dp (*python*). On all benchmarks, Pus is *at least* $2\times$ faster than both Wp and Dp. The result indicates that Pus has a great potential to be adopted widely as the computing power becomes stronger and stronger and more precise pointer analysis is desired in the future.

The reason why Pus is able to significantly outperform the state-of-the-art algorithms, especially when solving context-sensitive pointer analysis, is still rooted in our key insight that constraint graphs are *sparsely connected*. This property becomes even more essential in context-sensitive pointer analysis as one node in context-insensitive pointer analysis can correspond to multiple nodes in context-sensitive pointer analysis because the same variable is now analyzed separately under different contexts. This makes the constraint graph *sparser*. Thus, the effect brought by the update of one node becomes more *local* as it can only affect nodes under some particular contexts whereas one node can affect many neighbors in context-insensitive pointer analysis, even when the neighboring nodes represent variables in a mismatched context.

Despite all kinds of optimizations made when designing Pus, we still found it challenging to run context-sensitive pointer analysis on extremely large benchmarks using only commodity hardware. As the k-limiting for context-sensitive pointer analysis increases, the complexity of the algorithm and the size of the constraint graph grows exponentially, which makes the algorithm hard to scale on large benchmarks. During the experiments, we observed that more than 5 million nodes were created in the constraint graph for *PostgreSQL* in the first 5 minutes, which rapidly drains the memory of our machine. A more powerful machine is needed for evaluating Pus on *PostgreSQL*.

6 RELATED WORK

Pointer analysis is a fundamental building block for static program analyses and compiler optimizations. As such, precise and scalable pointer analysis has long been sought after as a holy grail that might unlock all secrets in the program analysis universe. Unfortunately, such a pointer analysis algorithm has yet to be found. All existing implementation of pointer analysis must make some trade off.

Across the decades, Andersen's inclusion based pointer analysis has emerged as the most popular pointer analyses [1]. Many works have been proposed to improve the base Andersen's analysis. Most of the previous research abstracts pointer analysis as a constraint graph and propagates the points-to information until a global fixed point. Heintze et al. [11] introduced a way to avoid the cost of computing the full transitive closure of the constraint graph. Instead a dynamic transitive closure is computed on demand and graph reachability queries are used to resolve points-to sets. As a result, cycle detection is achieved essentially for free as a result of the graph reachability queries. However this technique also introduces the potential for redundant work across reachability queries. Later works [10, 21, 22] topologically sort the constraint graph to reduce redundant points-to set propagation. Pereira et al. [23] proposed a new constraint solving algorithm, wave propagation, by separating the algorithm into three phases; collapsing of cycles, points-to propagation and insertion of new edges. These three phases are performed as a wave and repeated until a fixed point is reached. Pus advances the state-of-the-art by performing SCC detection and points-to set propagation on the *causality subgraph*, thus avoiding redundant computation in each iteration.

As the difficulty in developing an efficient constraint solving algorithm remains, researchers recently turned their attention to tackle the problem at new angles. D4 [18] first introduced an incremental algorithm for inclusion-based pointer analysis to enable differential pointer analysis on code changes. The algorithm of D4 is orthogonal to ours and Pus can be efficiently integrated with D4 to speed up its bootstrapping constraint solving process. DEA [15] introduced a faster algorithm to deal with positive-weight cycle in field-sensitive pointer analysis, while it still relies on wave propagation to compute the fixed point.

Another line of research formulates pointer analysis as a CFL-reachability problem. Reps et al. [25] modelled the flow-insensitive pointer analysis into a CFL-reachability problem. Spath et al. [29] proposed a flow- and context-sensitive demand-driven pointer analysis that models the pointer analysis as an IFDS problem, which then can be solved by CFL-reachability. This line of research is orthogonal to Pus, and Pus is more efficient when applications frequently query pointer analysis results.

Graph simplification techniques can be applied to both constraint-graph-based and CFL-reachability-based approaches to improve their scalability. Fahndrich et al. [7] first showed that collapsing SCC components in the constraint graph can significantly improve the performance of inclusion based pointer analysis. Pearce et al. [21] introduced an algorithm for online cycle detection. By keeping the constraint graph topologically sorted, cycle detection need only be run when a new edge violates the existing topological ordering. Since detecting cycles upon edge insertion was proven to be too costly, Pearce et al. [22] introduced an efficient field sensitive PTA that occasionally checks for and collapses cycles in the constraint graph. Hardekopf et al. [10] introduced Lazy Cycle Detection (LCD) and Hybrid Cycle Detection (HCD). LCD reduces runtime overhead even further by selectively triggering cycle detection only when identical points-to sets are discovered during transitive closure computation. HCD introduces an offline linear-time graph preprocessing stage that allows the online pointer analysis to detect cycles without the need for graph traversal at all. Pus extends the above techniques by not only applying general graph optimization techniques but also leveraging unique properties of constraint graph to only performing the SCC detection on causality subgraph. Thus, Pus can dynamically prune off the most ineffective edges to avoid redundant points-to set propagation.

Recent work by Li et al. [17] proposed to simplify the input labeled graph in a CFL-reachability problem by eliminating useless graph edges. Pus is similar to this work from a very high-level. However, this work primarily optimize the labeled graph in CFL-reachability problems while Pus focuses on simplifying the constraint graphs in pointer analysis.

Besides improving the solving algorithm, researchers have also proposed to use Datalog [5, 27, 28] for fast and easy pointer analysis implementation. While the experimental result indicates a great potential along the direction, fully customized pointer analysis solvers are still desired and used by many of the most recent works [18, 26, 32] as they are easier to be extended and tailored for different needs.

7 CONCLUSION

We have presented Partial Update Solver (Pus), a new constraint solving algorithm for inclusion-based pointer analysis. Pus significantly advances the state-of-the-art in reducing the time complexity by a quadratic factor. The key insight is that only a small portion of the constraint graph is effective for the points-to set propagation, which can be extracted efficiently into a subgraph, called *causality subgraph*. We have formally proved the correctness of Pus and extensively evaluated the performance of Pus on a wide range of real-world large complex programs. Our experimental results indicate that Pus is high scalable and significantly more efficient than the state-of-the-art Wp/Dp algorithms. Pus achieves more than 7× (2×) speedups when comparing to Wp/Dp in solving context-sensitive and context-insensitive pointer analyses respectively.

REFERENCES

[1] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language.* Ph.D. Dissertation. University of Cophenhagen.

[2] Mohamad Barbar, Yulei Sui, and Shiping Chen. 2020. Flow-sensitive type-based heap cloning. In *34th European Conference on Object-Oriented Programming (ECOOP 2020).* Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[3] Mohamad Barbar, Yulei Sui, and Shiping Chen. 2021. Object Versioning for Flow-Sensitive Pointer Analysis. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).* IEEE, 222–235.

[4] Rastisalv Bodík and Sadun Anik. 1998. Path-sensitive value-flow analysis. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* 237–251.

[5] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications.* 243–262.

[6] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. 2007. Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 480–491.

[7] Manuel Fähndrich, Jeffrey S Foster, Zhendong Su, and Alexander Aiken. 1998. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation.* 85–96.

[8] Gang Fan, Rongxin Wu, Qingkai Shi, Xiao Xiao, Jinguo Zhou, and Charles Zhang. 2019. Smoke: scalable path-sensitive memory leak detection for millions of lines of code. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE).* IEEE, 72–82.

[9] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. 2020. Temporal system call specialization for attack surface reduction. In *29th {USENIX} Security Symposium ({USENIX} Security 20).* 1749–1766.

[10] Ben Hardekopf and Calvin Lin. 2007. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 290–299.

[11] Nevin Heintze and Olivier Tardieu. 2001. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. *ACM SIGPLAN Notices* 36, 5 (2001), 254–263.

[12] Susan Horwitz. 1997. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19, 1 (1997), 1–6.

[13] Vineet Kahlon, Yu Yang, Sriram Sankaranarayanan, and Aarti Gupta. 2007. Fast and accurate static data-race detection for concurrent programs. In *International Conference on Computer Aided Verification.* Springer, 226–239.

[14] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86.

[15] Yuxiang Lei and Yulei Sui. 2019. Fast and precise handling of positive weight cycles for field-sensitive pointer analysis. In *International Static Analysis Symposium.* Springer, 27–47.

[16] Yanze Li, Bozhen Liu, and Jeff Huang. 2019. Sword: A scalable whole program race detector for java. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion).* IEEE, 75–78.

[17] Yuanbo Li, Qirun Zhang, and Thomas Reps. 2020. Fast graph simplification for interleaved Dyck-reachability. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation.* 780–793.

[18] Bozhen Liu and Jeff Huang. 2018. D4: Fast Concurrency Debugging with Parallel Differential Analysis. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018).* Association for Computing Machinery, New York, NY, USA, 359–373. https://doi.org/10.1145/3192366.3192390

[19] Bozhen Liu, Peiming Liu, Yanze Li, Chia-Che Tsai, Dilma Da Silva, and Jeff Huang. 2021. 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.* 725–739.

[20] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 308–319.

[21] David J Pearce, Paul HJ Kelly, and Chris Hankin. 2003. Online cycle detection and difference propagation for pointer analysis. In *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation.* IEEE, 3–12.

[22] David J Pearce, Paul HJ Kelly, and Chris Hankin. 2007. Efficient field-sensitive pointer analysis of C. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30, 1 (2007), 4–es.

[23] Fernando Magno Quintao Pereira and Daniel Berlin. 2009. Wave propagation and deep propagation for pointer analysis. In *2009 International Symposium on Code Generation and Optimization.* IEEE, 126–135.

[24] Ganesan Ramalingam. 1994. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 5 (1994), 1467–1471.

[25] Thomas Reps. 1998. Program analysis via graph reachability. *Information and software technology* 40, 11-12 (1998), 701–726.

[26] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: Fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 693–706.

[27] Yannis Smaragdakis and George Balatsouras. 2015. Pointer analysis. *Foundations and Trends in Programming Languages* 2, 1 (2015), 1–69.

[28] Yannis Smaragdakis and Martin Bravenboer. 2010. Using Datalog for fast and easy program analysis. In *International Datalog 2.0 Workshop.* Springer, 245–251.

[29] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-driven flow-and context-sensitive pointer analysis for java. In *30th European Conference on Object-Oriented Programming (ECOOP 2016).* Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[30] Manu Sridharan and Stephen J Fink. 2009. The complexity of Andersen's analysis in practice. In *International Static Analysis Symposium.* Springer, 205–221.

[31] Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* 32–41.

[32] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction.* 265–266.

[33] Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis.* 254–264.

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

[35] Yichen Xie and Alex Aiken. 2005. Context-and path-sensitive memory leak detection. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering.* 115–125.