Optimal and Perfectly Parallel Algorithms for On-demand Data-flow Analysis*

Krishnendu Chatterjee[1], Amir Kafshdar Goharshady[1], Rasmus Ibsen-Jensen[2], and Andreas Pavlogiannis[3]

[1] IST Austria, Klosterneuburg, Austria
[krishnendu.chatterjee, amir.goharshady]@ist.ac.at
[2] University of Liverpool, Liverpool, United Kingdom
r.ibsen-jensen@liverpool.ac.uk
[3] Aarhus University, Aarhus, Denmark
pavlogiannis@cs.au.dk

Abstract. Interprocedural data-flow analyses form an expressive and useful paradigm of numerous static analysis applications, such as live variables analysis, alias analysis and null pointers analysis. The most widely-used framework for interprocedural data-flow analysis is *IFDS*, which encompasses distributive data-flow functions over a finite domain. *On-demand* data-flow analyses restrict the focus of the analysis on specific program locations and data facts. This setting provides a natural split between (i) an *offline (or preprocessing) phase*, where the program is partially analyzed and analysis summaries are created, and (ii) an *online (or query) phase*, where analysis queries arrive on demand and the summaries are used to speed up answering queries.

In this work, we consider on-demand IFDS analyses where the queries concern program locations of the same procedure (aka same-context queries). We exploit the fact that flow graphs of programs have low treewidth to develop faster algorithms that are *space and time optimal* for many common data-flow analyses, in both the preprocessing and the query phase. We also use treewidth to develop query solutions that are *embarrassingly parallelizable*, i.e. the total work for answering each query is split to a number of threads such that each thread performs only a constant amount of work. Finally, we implement a static analyzer based on our algorithms, and perform a series of on-demand analysis experiments on standard benchmarks. Our experimental results show a drastic speed-up of the queries after only a lightweight preprocessing phase, which significantly outperforms existing techniques.

Keywords: Data-flow analysis, IFDS, Treewidth

*The research was partly supported by Austrian Science Fund (FWF) Grant No. NFN S11407-N23 (RiSE/SHiNE), FWF Schrödinger Grant No. J-4220, Vienna Science and Technology Fund (WWTF) Project ICT15-003, Facebook PhD Fellowship Program, IBM PhD Fellowship Program, and DOC Fellowship No. 24956 of the Austrian Academy of Sciences (ÖAW). A longer version of this work is available at [17].

P. Müller (Ed.): ESOP 2020, LNCS 12075, pp. 112–140, 2020.
https://doi.org/10.1007/978-3-030-44914-8_5

1 Introduction

Static data-flow analysis. Static program analysis is a fundamental approach for both analyzing program correctness and performing compiler optimizations [25,39,44,64,30]. Static data-flow analyses associate with each program location a set of data-flow facts which are guaranteed to hold under all program executions, and these facts are then used to reason about program correctness, report erroneous behavior, and optimize program execution. Static data-flow analyses have numerous applications, such as in pointer analysis (e.g., points-to analysis and detection of null pointer dereferencing) [46,57,61,62,66,67,69], in detecting privacy and security issues (e.g., taint analysis, SQL injection analysis) [3,37,31,33,47,40], as well as in compiler optimizations (e.g., constant propagation, reaching definitions, register allocation) [50,32,55,13,2].

Interprocedural analysis and the IFDS framework. Data-flow analyses fall in two large classes: *intraprocedural* and *interprocedural*. In the former, each procedure of the program is analyzed in isolation, ignoring the interaction between procedures which occurs due to parameter passing/return. In the latter, all procedures of the program are analyzed together, accounting for such interactions, which leads to results of increased precision, and hence is often preferable to intraprocedural analysis [49,54,59,60]. To filter out false results, interprocedural analyses typically employ call-context sensitivity, which ensures that the underlying execution paths respect the calling context of procedure invocations. One of the most widely used frameworks for interprocedural data-flow analysis is the framework of Interprocedural Finite Distributive Subset (IFDS) problems [50], which offers a unified formulation of a wide class of interprocedural data-flow analyses as a reachability problem. This elegant algorithmic formulation of data-flow analysis has been a topic of active study, allowing various subsequent practical improvements [36,45,8,3,47,56] and implementations in prominent static analysis tools such as Soot [7] and WALA [1].

On-demand analysis. Exhaustive data-flow analysis is computationally expensive and often unnecessary. Hence, a topic of great interest in the community is that of *on-demand* data-flow analysis [4,27,36,51,48,68,45]. On-demand analyses have several applications, such as (quoting from [36,48]) (i) narrowing down the focus to specific points of interest, (ii) narrowing down the focus to specific data-flow facts of interest, (iii) reducing work in preliminary phases, (iv) sidestepping incremental updating problems, and (v) offering demand analysis as a user-level operation. On-demand analysis is also extremely useful for speculative optimizations in just-in-time compilers [24,43,5,29], where dynamic information can dramatically increase the precision of the analysis. In this setting, it is crucial that the the on-demand analysis runs fast, to incur as little overhead as possible.

Example 1. As a toy motivating example, consider the partial program shown in Figure 1, compiled with a just-in-time compiler that uses speculative optimizations. Whether the compiler must compile the expensive function `h` depends on whether x is null in line 6. Performing a null-pointer analysis from the entry of

```
1 void f(int b){                    9 void g(int *&x, int *y){
2   int *x = NULL, *y = NULL;      10   x=y;
3   if(b > 1)                      11 }
4     y = &b;
5   g(x,y);                        12 void h(){
6   if(x==NULL)                    13   //An expensive
7     h();                         14   //function
8 }                                15 }
```

Fig. 1: A partial C++ program.

`f` reveals that x might be null in line 6. Hence, if the decision to compile `h` relies only on an offline static analysis, `h` is always compiled, even when not needed.

Now consider the case where the execution of the program is in line 4, and at this point the compiler decides on whether to compile `h`. It is clear that given this information, x cannot be null in line 6 and thus `h` does not have to be compiled. As we have seen above, this decision can not be made based on offline analysis. On the other hand, an *on-demand* analysis starting from the current program location will correctly conclude that x is not null in line 6. Note however, that this decision is made by the compiler during runtime. Hence, such an on-demand analysis is useful only if it can be performed extremely fast. It is also highly desirable that the time for running this analysis is predictable, so that the compiler can decide whether to run the analysis or simply compile `h` proactively.

The techniques we develop in this paper answer the above challenges rigorously. Our approach exploits a key structural property of flow graphs of programs, called treewidth.

Treewidth of programs. A very well-studied notion in graph theory is the concept of *treewidth* of a graph, which is a measure of how similar a graph is to a tree (a graph has treewidth 1 precisely if it is a tree) [52]. On one hand the treewidth property provides a mathematically elegant way to study graphs, and on the other hand there are many classes of graphs which arise in practice and have constant treewidth. The most important example is that the flow graph for `goto`-free programs in many classic programming languages have constant treewidth [63]. The low treewidth of flow graphs has also been confirmed experimentally for programs written in Java [34], C [38], Ada [12] and Solidity [15].

Treewidth has important algorithmic implications, as many graph problems that are hard to solve in general admit efficient solutions on graphs of low treewidth. In the context of program analysis, this property has been exploited to develop improvements for register allocation [63,9] (a technique implemented in the Small Device C Compiler [28]), cache management [18], on-demand algebraic path analysis [16], on-demand *intraprocedural* data-flow analysis of concurrent programs [20] and data-dependence analysis [14].

Problem statement. We focus on on-demand data-flow analysis in IFDS [50,36,48]. The input consists of a supergraph G of n vertices, a data-fact domain D and a data-flow transformer function M. Edges of G capture control-flow within each procedure, as well as procedure invocations and returns. The set D defines the domain of the analysis, and contains the data facts to be discovered by the analysis for each program location. The function M associates with every edge (u, v) of G a data-flow transformer $M(u, v) : 2^D \to 2^D$. In words, $M(u, v)$ defines the set of data facts that hold at v in some execution that transitions from u to v, given the set of data facts that hold at u.

On-demand analysis brings a natural separation between (i) an *offline (or preprocessing) phase*, where the program is partially analyzed, and (ii) an *online (or query) phase*, where on-demand queries are handled. The task is to preprocess the input in the offline phase, so that in the online phase, the following types of on-demand queries are answered efficiently:

1. A *pair query* has the form (u, d_1, v, d_2), where u, v are vertices of G in the same procedure, and d_1, d_2 are data facts. The goal is to decide if there exists an execution that starts in u and ends in v, and given that the data fact d_1 held at the beginning of the execution, the data fact d_2 holds at the end. These are known as *same-context* queries and are very common in data-flow analysis [23,50,16].
2. A *single-source* query has the form (u, d_1), where u is a vertex of G and d_1 is a data fact. The goal is to compute for every vertex v that belongs to the same procedure as u, all the data facts that might hold in v as witnessed by executions that start in u and assuming that d_1 holds at the beginning of each such execution.

Previous results. The on-demand analysis problem admits a number of solutions that lie in the preprocessing/query spectrum. On the one end, the preprocessing phase can be disregarded, and every on-demand query be treated anew. Since each query starts a separate instance of IFDS, the time to answer it is $O(n \cdot |D|^3)$, for both pair and single-source queries [50]. On the other end, all possible queries can be pre-computed and cached in the preprocessing phase in time $O(n^2 \cdot |D|^3)$, after which each query costs time proportional to the size of the output (i.e., $O(1)$) for pair queries and $O(n \cdot |D|)$ for single-source queries). Note that this full preprocessing also incurs a cost $O(n^2 \cdot |D|^2)$ in space for storing the cache table, which is often prohibitive. On-demand analysis was more thoroughly studied in [36]. The main idea is that, instead of pre-computing the answer to all possible queries, the analysis results obtained by handling each query are memoized to a cache table, and are used for speeding up the computation of subsequent queries. This is a heuristic-based approach that often works well in practice, however, the only guarantee provided is that of *same-worst-case-complexity*, which states that in the worst case, the algorithm uses $O(n^2 \cdot |D|^3)$ time and $O(n^2 \cdot |D|^2)$ space, similarly to the complete preprocessing case. This guarantee is inadequate for runtime applications such as the example of Figure 1, as it would require either (i) to run a full analysis, or (ii) to run a partial analysis which might wrongly conclude that `h` is reachable, and thus compile it. Both cases incur a

large runtime overhead, either because we run a full analysis, or because we compile an expensive function.

Our contributions. We develop algorithms for on-demand IFDS analyses that have strong worst-case time complexity guarantees and thus lead to more predictable performance than mere heuristics. The contributions of this work are as follows:

1. We develop an algorithm that, given a program represented as a supergraph of size n and a data fact domain D, solves the on-demand same-context IFDS problem while spending (i) $O(n \cdot |D|^3)$ time in the preprocessing phase, and (ii) $O(\lceil |D|/\log n \rceil)$ time for a pair query and $O(n \cdot |D|^2/\log n)$ time for a single-source query in the query phase. Observe that when $|D| = O(1)$, the preprocessing and query times are proportional to the size of the input and outputs, respectively, and are thus *optimal*[§]. In addition, our algorithm uses $O(n \cdot |D|^2)$ space at all times, which is proportional to the size of the input, and is thus *space optimal*. Hence, our algorithm not only improves upon previous state-of-the-art solutions, but also ensures optimality in both time and space.
2. We also show that after our one-time preprocessing, each query is *embarrassingly parallelizable*, i.e., every bit of the output can be produced by a single thread in $O(1)$ time. This makes our techniques particularly useful to speculative optimizations, since the analysis is guaranteed to take constant time and thus incur little runtime overhead. Although the parallelization of data-flow analysis has been considered before [41,42,53], this is the first time to obtain solutions that span beyond heuristics and offer theoretical guarantees. Moreover, this is a rather surprising result, given that general IFDS is known to be P-complete.
3. We implement our algorithms on a static analyzer and experimentally evaluate their performance on various static analysis clients over a standard set of benchmarks. Our experimental results show that after only a lightweight preprocessing, we obtain a significant speedup in the query phase compared to standard on-demand techniques in the literature. Also, our parallel implementation achieves a speedup close to the theoretical optimal, which illustrates that the perfect parallelization of the problem is realized by our approach in practice.

Recently, we exploited the low-treewidth property of programs to obtain faster algorithms for algebraic path analysis [16] and intraprocedural reachability [21]. Data-flow analysis can be reduced to these problems. Hence, the algorithms in [16,21] can also be applied to our setting. However, our new approach has two important advantages: (i) we show how to answer queries in a perfectly parallel manner, and (ii) reducing the problem to algebraic path properties and then applying the algorithms in [16,21] yields $O(n \cdot |D|^3)$ preprocessing time and $O(n \cdot \log n \cdot |D|^2)$ space, and has pair and single-source query time $O(|D|)$ and $O(n \cdot |D|^2)$. Hence, our space usage and query times are better by a factor of

[§]Note that we count the input itself as part of the space usage.

$\log n$[¶]. Moreover, when considering the complexity wrt n, i.e. considering D to be a constant, these results are optimal wrt both time and space. Hence, no further improvement is possible.

Remark. Note that our approach does not apply to arbitrary CFL reachability in constant treewidth. In addition to the treewidth, our algorithms also exploit specific structural properties of IFDS. In general, small treewidth alone does not improve the complexity of CFL reachability [14].

2 Preliminaries

Model of computation. We consider the standard RAM model with word size $W = \Theta(\log n)$, where n is the size of our input. In this model, one can store W bits in one word (aka "word tricks") and arithmetic and bitwise operations between pairs of words can be performed in $O(1)$ time. In practice, word size is a property of the machine and not the analysis. Modern machines have words of size at least 64. Since the size of real-world input instances never exceeds 2^{64}, the assumption of word size $W = \Theta(\log n)$ is well-realized in practice and no additional effort is required by the implementer to account for W in the context of data flow analysis.

Graphs. We consider directed graphs $G = (V, E)$ where V is a finite set of vertices and $E \subseteq V \times V$ is a set of directed edges. We use the term graph to refer to directed graphs and will explicitly mention if a graph is undirected. For two vertices $u, v \in V$, a path P from u to v is a finite sequence of vertices $P = (w_i)_{i=0}^{k}$ such that $w_0 = u$, $w_k = v$ and for every $i < k$, there is an edge from w_i to w_{i+1} in E. The length $|P|$ of the path P is equal to k. In particular, for every vertex u, there is a path of length 0 from u to itself. We write $P : u \rightsquigarrow v$ to denote that P is a path from u to v and $u \rightsquigarrow v$ to denote the existence of such a path, i.e. that v is reachable from u. Given a set $V' \subseteq V$ of vertices, the induced subgraph of G on V' is defined as $G[V'] = (V', E \cap (V' \times V'))$. Finally, the graph G is called *bipartite* if the set V can be partitioned into two sets V_1, V_2, so that every edge has one end in V_1 and the other in V_2, i.e. $E \subseteq (V_1 \times V_2) \cup (V_2 \times V_1)$.

2.1 The IFDS Framework

IFDS [50] is a ubiquitous and general framework for interprocedural data-flow analyses that have finite domains and distributive flow functions. It encompasses a wide variety of analyses, including truly-live variables, copy constant propagation, possibly-uninitialized variables, secure information-flow, and gen/kill or bitvector problems such as reaching definitions, available expressions and live variables [50,7]. IFDS obtains *interprocedurally precise* solutions. In contrast to intraprocedural analysis, in which precise denotes "meet-over-all-paths", interprocedurally precise solutions only consider valid paths, i.e. paths in which when

[¶]This improvement is due to the differences in the preprocessing phase. Our algorithms for the query phase are almost identical to our previous work.

a function reaches its end, control returns back to the site of the most recent call [58].

Flow graphs and supergraphs. In IFDS, a program with k procedures is specified by a *supergraph*, i.e. a graph $G = (V, E)$ consisting of k flow graphs $G_1, \ldots, G_k$, one for each procedure, and extra edges modeling procedure-calls. Flow graphs represent procedures in the usual way, i.e. they contain one vertex v_i for each statement i and there is an edge from v_i to v_j if the statement j may immediately follow the statement i in an execution of the procedure. The only exception is that a procedure-call statement i is represented by two vertices, a *call* vertex c_i and a *return-site* vertex r_i. The vertex c_i only has incoming edges, and the vertex r_i only has outgoing edges. There is also a *call-to-return-site* edge from c_i to r_i. The call-to-return-site edges are included for passing intraprocedural information, such as information about local variables, from c_i to r_i. Moreover, each flow graph G_l has a unique *start* vertex s_l and a unique *exit* vertex e_l.

The supergraph G also contains the following edges for each procedure-call i with call vertex c_i and return-site vertex r_i that calls a procedure l: (i) an interprocedural *call-to-start* edge from c_i to the start vertex of the called procedure, i.e. s_l, and (ii) an interprocedural *exit-to-return-site* edge from the exit vertex of the called procedure, i.e. e_l, to r_i.

Example 2. Figure 2 shows a simple C++ program on the left and its supergraph on the right. Each statement i of the program has a corresponding vertex v_i in the supergraph, except for statement 7, which is a procedure-call statement and hence has a corresponding call vertex c_7 and return-site vertex r_7.

```
1  void f(int *&x, int *y){
2    y = new int(1);
3    y = new int(2);
4  }

5  int main(){
6    int *x, *y;
7    f(x,y);
8    *x += *y;
9  }
```

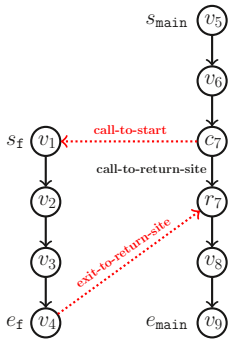

Fig. 2: A C++ program (left) and its supergraph (right).

Interprocedurally valid paths. Not every path in the supergraph G can potentially be realized by an execution of the program. Consider a path P in G and let P' be the sequence of vertices obtained by removing every v_i from P, i.e. P' only consists of c_i's and r_i's. Then, P is called a *same-context valid path* if P' can be generated from S in this grammar:

$$\begin{array}{ll} S & \to c_i \;\; S \;\; r_i \;\; S \;\; \text{for a procedure-call statement } i \\ \mid & \varepsilon \end{array}.$$

Moreover, P is called an *interprocedurally valid path* or simply *valid* if P' can be generated from the nonterminal S' in the following grammar:

$$\begin{array}{ll} S' & \to S' \;\; c_i \;\; S \quad \text{for a procedure-call statement } i \\ \mid & S \end{array}.$$

For any two vertices u, v of the supergraph G, we denote the set of all interprocedurally valid paths from u to v by $\mathsf{IVP}(u, v)$ and the set of all same-context valid paths from u to v by $\mathsf{SCVP}(u, v)$. Informally, a valid path starts from a statement in a procedure p of the program and goes through a number of procedure-calls while respecting the rule that whenever a procedure ends, control should return to the return-site in its parent procedure. A same-context valid path is a valid path in which every procedure-call ends and hence control returns back to the initial procedure p in the same context.

IFDS [50]. An IFDS problem *instance* is a tuple $I = (G, D, F, M, \sqcap)$ where:

- $G = (V, E)$ is a supergraph as above.
- D is a finite set, called the *domain*, and each $d \in D$ is called a *data flow fact*.
- The *meet operator* $\sqcap$ is either intersection or union.
- $F \subseteq 2^D \to 2^D$ is a set of *distributive flow functions* over $\sqcap$, i.e. for each function $f \in F$ and every two sets of facts $D_1, D_2 \subseteq D$, we have $f(D_1 \sqcap D_2) = f(D_1) \sqcap f(D_2)$.
- $M : E \to F$ is a map that assigns a distributive flow function to each edge of the supergraph.

Let $P = (w_i)_{i=0}^k$ be a path in G, $e_i = (w_{i-1}, w_i)$ and $m_i = M(e_i)$. In other words, the e_i's are the edges appearing in P and the m_i's are their corresponding distributive flow functions. The *path function* of P is defined as: $\mathsf{pf}_P := m_k \circ \cdots \circ m_2 \circ m_1$ where $\circ$ denotes function composition. The solution of I is the collection of values $\{\mathsf{MVP}_v\}_{v \in V}$:

$$\mathsf{MVP}_v := \bigsqcap_{P \in \mathsf{IVP}(s_{\mathsf{main}}, v)} \mathsf{pf}_P(D).$$

Intuitively, the solution is defined by taking *meet-over-all-valid-paths*. If the meet operator is union, then MVP_v is the set of data flow facts that *may* hold at v, when v is reached in *some* execution of the program. Conversely, if the meet operator is intersection, then MVP_v consists of data flow facts that *must* hold at v in *every* execution of the program that reaches v. Similarly, we define the same-context solution of I as the collection of values $\{\mathsf{MSCP}_v\}_{v \in V_{\mathsf{main}}}$ defined as follows:

$$\mathsf{MSCP}_v := \bigsqcap_{P \in \mathsf{SCVP}(s_{\mathsf{main}}, v)} \mathsf{pf}_P(D). \tag{1}$$

The intuition behind MSCP is similar to that of MVP, except that in MSCP_v we consider *meet-over-same-context-paths* (corresponding to runs that return to the same stack state).

Remark 1. We note two points about the IFDS framework:
- As in [50], we only consider IFDS instances in which the meet operator is union. Instances with intersection can be reduced to union instances by dualization [50].
- For brevity, we are considering a global domain D, while in many applications the domain is procedure-specific. This does not affect the generality of our approach and our algorithms remain correct for the general case where each procedure has its own dedicated domain. Indeed, our implementation supports the general case.

Succinct representations. A distributive function $f : 2^D \to 2^D$ can be succinctly represented by a relation $R_f \subseteq (D \cup \{\mathbf{0}\}) \times (D \cup \{\mathbf{0}\})$ defined as:

$$\begin{aligned} R_f := & \{(\mathbf{0}, \mathbf{0})\} \\ & \cup \ \{(\mathbf{0}, b) \mid b \in f(\emptyset)\} \\ & \cup \ \{(a, b) \mid b \in f(\{a\}) - f(\emptyset)\}. \end{aligned}$$

Given that f is distributive over union, we have $f(\{d_1, \ldots, d_k\}) = f(\{d_1\}) \cup \cdots \cup f(\{d_k\})$. Hence, to specify f it is sufficient to specify $f(\emptyset)$ and $f(\{d\})$ for each $d \in D$. This is exactly what R_f does. In short, we have: $f(\emptyset) = \{b \in D \mid (\mathbf{0}, b) \in R_f\}$ and $f(\{d\}) = f(\emptyset) \cup \{b \in D \mid (d, b) \in R_f\}$. Moreover, we can represent the relation R_f as a bipartite graph H_f in which each part consists of the vertices $D \cup \{\mathbf{0}\}$ and R_f is the set of edges. For brevity, we define $D^* := D \cup \{\mathbf{0}\}$.

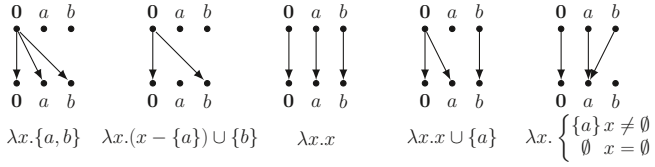

Fig. 3: Succinct representation of several distributive functions.

Example 3. Let $D = \{a, b\}$. Figure 3 provides several examples of bipartite graphs representing distributive functions.

Bounded Bandwidth Assumption. Following [50], we assume that the bandwidth in function calls and returns is bounded by a constant. In other words, there is a small constant b, such that for every edge e that is a call-to-start or exit-to-return-site edge, every vertex in the graph representation $H_{M(e)}$ has degree b or less. This is a classical assumption in IFDS [50,7] and models the fact that every parameter in a called function is only dependent on a few variables in the callee (and conversely, every returned value is only dependent on a few variables in the called function).

Composition of distributive functions. Let f and g be distributive functions and R_f and R_g their succinct representations. It is easy to verify that $g \circ f$ is also distributive, hence it has a succinct representation $R_{g \circ f}$. Moreover, we have $R_{g \circ f} = R_f ; R_g = \{(a, b) \mid \exists c \quad (a, c) \in R_f \wedge (c, b) \in R_g\}$.

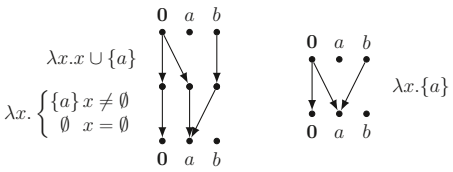

Fig. 4: Obtaining $H_{g \circ f}$ (right) from H_f and H_g (left)

Example 4. In terms of graphs, to compute $H_{g \circ f}$, we first take H_f and H_g, then contract corresponding vertices in the lower part of H_f and the upper part of H_g, and finally compute reachability from the topmost part to the bottommost part of the resulting graph. Consider $f(x) = x \cup \{a\}$, $g(x) = \{a\}$ for $x \neq \emptyset$ and $g(\emptyset) = \emptyset$, then $g \circ f(x) = \{a\}$ for all $x \subseteq D$. Figure 4 shows contracting of corresponding vertices in H_f and H_g (left) and using reachability to obtain $H_{g \circ f}$ (right).

Exploded supergraph. Given an IFDS instance $I = (G, D, F, M, \cup)$ with supergraph $G = (V, E)$, its *exploded supergraph* $\overline{G}$ is obtained by taking $|D^*|$ copies of each vertex in V, one corresponding to each element of D^*, and replacing each edge e with the graph representation $H_{M(e)}$ of the flow function $M(e)$. Formally, $\overline{G} = (\overline{V}, \overline{E})$ where $\overline{V} = V \times D^*$ and

$$\overline{E} = \left\{((u, d_1), (v, d_2)) \quad \mid \quad e = (u, v) \in E \ \wedge \ (d_1, d_2) \in R_{M(e)}\right\}.$$

A path $\overline{P}$ in $\overline{G}$ is (same-context) valid, if the path P in G, obtained by ignoring the second component of every vertex in $\overline{P}$, is (same-context) valid. As shown in [50], for a data flow fact $d \in D$ and a vertex $v \in V$, we have $d \in \mathsf{MVP}_v$ iff there is a valid path in $\overline{G}$ from (s_{main}, d') to (v, d) for some $d' \in D \cup \{\mathbf{0}\}$. Hence, the IFDS problem is reduced to reachability by valid paths in $\overline{G}$. Similarly, the same-context IFDS problem is reduced to reachability by same-context valid paths in $\overline{G}$.

Example 5. Consider a null pointer analysis on the program in Figure 2. At each program point, we want to know which pointers can potentially be null. We first model this problem as an IFDS instance. Let $D = \{\bar{x}, \bar{y}\}$, where $\bar{x}$ is the data flow fact that x might be null and $\bar{y}$ is defined similarly. Figure 5 shows the same program and its exploded supergraph.

At point 8, the values of both pointers x and y are used. Hence, if either of x or y is null at 8, a null pointer error will be raised. However, as evidenced by

the two valid paths shown in red, both x and y might be null at 8. The pointer y might be null because it is passed to the function `f` by value (instead of by reference) and keeps its local value in the transition from c_7 to r_7, hence the edge $((c_7, \bar{y}), (r_7, \bar{y}))$ is in $\overline{G}$. On the other hand, the function `f` only initializes y, which is its own local variable, and does not change x (which is shared with `main`).

```
1  void f(int *&x, int *y) {
2    y = new int(1);
3    y = new int(2);
4  }

5  int main() {
6    int *x, *y;
7    f(x,y);
8    *x += *y;
9  }
```

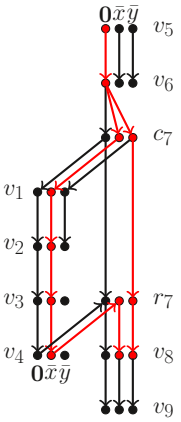

Fig. 5: A Program (left) and its Exploded Supergraph (right).

2.2 Trees and Tree Decompositions

Trees. A rooted tree $T = (V_T, E_T)$ is an undirected graph with a distinguished "root" vertex $r \in V_T$, in which there is a unique path P_v^u between every pair $\{u, v\}$ of vertices. We refer to the number of vertices in V_T as the *size* of T. For an arbitrary vertex $v \in V_T$, the *depth* of v, denoted by d_v, is defined as the length of the unique path $P_v^r : r \rightsquigarrow v$. The *depth* or *height* of T is the maximum depth among its vertices. A vertex u is called an *ancestor* of v if u appears in P_v^r. In this case, v is called a *descendant* of u. In particular, r is an ancestor of every vertex and each vertex is both an ancestor and a descendant of itself. We denote the set of ancestors of v by $\mathsf{A}_v^{\uparrow}$ and its descendants by $\mathsf{D}_v^{\downarrow}$. It is straightforward to see that for every $0 \leq d \leq \mathsf{d}_v$, the vertex v has a unique ancestor with depth d. We denote this ancestor by a_v^d. The ancestor $p_v = \mathsf{a}_v^{\mathsf{d}_v - 1}$ of v at depth $\mathsf{d}_v - 1$ is called the *parent* of v and v is a *child* of p_v. The subtree $T_v^{\downarrow}$ corresponding to v is defined as $T[\mathsf{D}_v^{\downarrow}] = (\mathsf{D}_v^{\downarrow}, E_T \cap 2^{\mathsf{D}_v^{\downarrow}})$, i.e. the part of T that consists of v and its descendants. Finally, a vertex $v \in V_T$ is called a *leaf* if it has no children. Given two vertices $u, v \in V_T$, the *lowest common ancestor* $\mathsf{lca}(u, v)$ of u and v is defined as $\mathrm{argmax}_{w \in \mathsf{A}_u^{\uparrow} \cap \mathsf{A}_v^{\uparrow}} \mathsf{d}_w$. In other words, $\mathsf{lca}(u, v)$ is the common ancestor of u and v with maximum depth, i.e. which is farthest from the root.

Lemma 1 ([35]). *Given a rooted tree T of size n, there is an algorithm that preprocesses T in $O(n)$ and can then answer lowest common ancestor queries, i.e. queries that provide two vertices u and v and ask for $\mathsf{lca}(u, v)$, in $O(1)$.*

Tree decompositions [52]. Given a graph $G = (V, E)$, a *tree decomposition* of G is a rooted tree $T = (\mathfrak{B}, E_T)$ such that:

(i) Each vertex $b \in \mathfrak{B}$ of T has an associated subset $V(b) \subseteq V$ of vertices of G and $\bigcup_{b \in \mathfrak{B}} V(b) = V$. For clarity, we call each vertex of T a "bag" and reserve the word vertex for G. Informally, each vertex must appear in some bag.

(ii) For all $(u, v) \in E$, there exists a bag $b \in \mathfrak{B}$ such that $u, v \in V(b)$, i.e. every edge should appear in some bag.

(iii) For any pair of bags $b_i, b_j \in \mathfrak{B}$ and any bag b_k that appears in the path $P : b_i \rightsquigarrow b_j$, we have $V(b_i) \cap V(b_j) \subseteq V(b_k)$, i.e. each vertex should appear in a connected subtree of T.

The *width* of the tree decomposition $T = (\mathfrak{B}, E_T)$ is defined as the size of its largest bag minus 1. The *treewidth* $\mathsf{tw}(G)$ of a graph G is the minimal width among its tree decompositions. A vertex $v \in V$ appears in a connected subtree, so there is a unique bag b with the smallest possible depth such that $v \in V(b)$. We call b the *root bag* of v and denote it by $\mathsf{rb}(v)$.

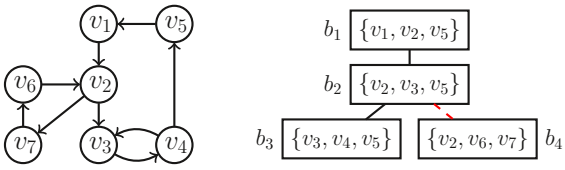

Fig. 6: A Graph G (left) and its Tree Decomposition T (right).

It is well-known that flow graphs of programs have typically small treewidth [63]. For example, programs written in Pascal, C, and Solidity have treewidth at most 3, 6 and 9, respectively. This property has also been confirmed experimentally for programs written in Java [34], C [38] and Ada [12]. The challenge is thus to exploit treewidth for faster interprocedural on-demand analyses. The first step in this approach is to compute tree decompositions of graphs. As the following lemma states, tree decompositions of low-treewidth graphs can be computed efficiently.

Lemma 2 ([11]). *Given a graph G with constant treewidth t, a binary tree decomposition of size $O(n)$ bags, height $O(\log n)$ and width $O(t)$ can be computed in linear time.*

Separators [26]. The key structural property that we exploit in low-treewidth flow graphs is a separation property. Let $A, B \subseteq V$. The pair (A, B) is called a *separation* of G if (i) $A \cup B = V$, and (ii) no edge connects a vertex in $A - B$

to a vertex in $B - A$ or vice versa. If (A, B) is a separation, the set $A \cap B$ is called a *separator*. The following lemma states such a separation property for low-treewidth graphs.

Lemma 3 (Cut Property [26]). *Let $T = (\mathfrak{B}, E_T)$ be a tree decomposition of $G = (V, E)$ and $e = \{b, b'\} \in E_T$. If we remove e, the tree T breaks into two connected components, T^b and $T^{b'}$, respectively containing b and b'. Let $A = \bigcup_{t \in T^b} V(t)$ and $B = \bigcup_{t \in T^{b'}} V(t)$. Then (A, B) is a separation of G and its corresponding separator is $A \cap B = V(b) \cap V(b')$.*

Example 6. Figure 6 shows a graph and one of its tree decompositions with width 2. In this example, we have $\mathsf{rb}(v_5) = b_1, \mathsf{rb}(v_3) = b_2, \mathsf{rb}(v_4) = b_3$, and $\mathsf{rb}(v_7) = b_4$. For the separator property of Lemma 3, consider the edge $\{b_2, b_4\}$. By removing it, T breaks into two parts, one containing the vertices $A = \{v_1, v_2, v_3, v_4, v_5\}$ and the other containing $B = \{v_2, v_6, v_7\}$. We have $A \cap B = \{v_2\} = V(b_2) \cap V(b_4)$. Also, any path from $B - A = \{v_6, v_7\}$ to $A - B = \{v_1, v_3, v_4, v_5\}$ or vice versa must pass through $\{v_2\}$. Hence, (A, B) is a separation of G with separator $V(b_2) \cap V(b_4) = \{v_2\}$.

3 Problem definition

We consider same-context IFDS problems in which the flow graphs G_i have a treewidth of at most t for a fixed constant t. We extend the classical notion of same-context IFDS solution in two ways: (i) we allow arbitrary start points for the analysis, i.e. we do not limit our analyses to same-context valid paths that start at s_{main}; and (ii) instead of a one-shot algorithm, we consider a two-phase process in which the algorithm first preprocesses the input instance and is then provided with a series of queries to answer. We formalize these points below. We fix an IFDS instance $I = (G, D, F, M, \cup)$ with exploded supergraph $\overline{G} = (\overline{V}, \overline{E})$.

Meet over same-context valid paths. We extend the definition of MSCP by specifying a start vertex u and an initial set Δ of data flow facts that hold at u. Formally, for any vertex v that is in the same flow graph as u, we define:

$$\mathsf{MSCP}_{u,\Delta,v} := \bigsqcap_{P \in \mathsf{SCVP}(u,v)} \mathsf{pf}_P(\Delta). \tag{2}$$

The only difference between (2) and (1) is that in (1), the start vertex u is fixed as s_{main} and the initial data-fact set Δ is fixed as D, while in (2), they are free to be any vertex/set.

Reduction to reachability. As explained in Section 2.1, computing MSCP is reduced to reachability via same-context valid paths in the exploded supergraph $\overline{G}$. This reduction does not depend on the start vertex and initial data flow facts. Hence, for a data flow fact $d \in D$, we have $d \in \mathsf{MSCP}_{u,\Delta,v}$ iff in the exploded supergraph $\overline{G}$ the vertex (v, d) is reachable via same-context valid paths from a vertex (u, δ) for some $\delta \in \Delta \cup \{\mathbf{0}\}$. Hence, we define the following types of queries:

Pair query. A pair query provides two vertices (u, d_1) and (v, d_2) of the exploded supergraph $\overline{G}$ and asks whether they are reachable by a same-context valid path. Hence, the answer to a pair query is a single bit. Intuitively, if $d_2 = \mathbf{0}$, then the query is simply asking if v is reachable from u by a same-context valid path in G. Otherwise, d_2 is a data flow fact and the query is asking whether $d_2 \in \mathsf{MSCP}_{u,\{d_1\}\cap D,v}$.

Single-source query. A single-source query provides a vertex (u, d_1) and asks for all vertices (v, d_2) that are reachable from (u, d_1) by a same-context valid path. Assuming that u is in the flow graph $G_i = (V_i, E_i)$, the answer to the single source query is a sequence of $|V_i| \cdot |D^*|$ bits, one for each $(v, d_2) \in V_i \times D^*$, signifying whether it is reachable by same-context valid paths from (u, d_1). Intuitively, a single-source query asks for all pairs (v, d_2) such that (i) v is reachable from u by a same-context valid path and (ii) $d_2 \in \mathsf{MSCP}_{u,\{d_1\}\cap D,v} \cup \{\mathbf{0}\}$.

Intuition. We note the intuition behind such queries. We observe that since the functions in F are distributive over $\cup$, we have $\mathsf{MSCP}_{u,\Delta,v} = \cup_{\delta\in\Delta}\mathsf{MSCP}_{u,\{\delta\},v}$, hence $\mathsf{MSCP}_{u,\Delta,v}$ can be computed by $O(|\Delta|)$ single-source queries.

4 Treewidth-based Data-flow Analysis

4.1 Preprocessing

The original solution to the IFDS problem, as first presented in [50], reduces the problem to reachability over a newly constructed graph. We follow a similar approach, except that we exploit the low-treewidth property of our flow graphs at every step. Our preprocessing is described below. It starts with computing constant-width tree decompositions for each of the flow graphs. We then use standard techniques to make sure that our tree decompositions have a nice form, i.e. that they are balanced and binary. Then comes a reduction to reachability, which is similar to [50]. Finally, we precompute specific useful reachability information between vertices in each bag and its ancestors. As it turns out in the next section, this information is sufficient for computing reachability between any pair of vertices, and hence for answering IFDS queries.

Overview. Our preprocessing consists of the following steps:

(1) **Finding Tree Decompositions.** In this step, we compute a tree decomposition $T_i = (\mathfrak{B}_i, E_{T_i})$ of constant width t for each flow graph G_i. This can either be done by applying the algorithm of [10] directly on G_i, or by using an algorithm due to Thorup [63] and parsing the program.

(2) **Balancing and Binarizing.** In this step, we balance the tree decompositions T_i using the algorithm of Lemma 2 and make them binary using the standard process of [22].

(3) **LCA Preprocessing.** We preprocess the T_i's for answering lowest common ancestor queries using Lemma 1.

(4) **Reduction to Reachability.** In this step, we modify the exploded supergraph $\overline{G} = (\overline{V}, \overline{E})$ to obtain a new graph $\hat{G} = (\overline{V}, \hat{E})$, such that for every pair of vertices (u, d_1) and (v, d_2), there is a path from (u, d_1) to (v, d_2) in

$\hat{G}$ iff there is a *same-context valid path* from (u, d_1) to (v, d_2) in $\overline{G}$. So, this step reduces the problem of reachability via same-context valid paths in $\overline{G}$ to simple reachability in $\hat{G}$.

(5) **Local Preprocessing.** In this step, for each pair of vertices (u, d_1) and (v, d_2) for which there exists a bag b such that both u and v appear in b, we compute and cache whether $(u, d_1) \rightsquigarrow (v, d_2)$ in $\hat{G}$. We write $(u, d_1) \rightsquigarrow_{\mathsf{local}} (v, d_2)$ to denote a reachability established in this step.

(6) **Ancestors Reachability Preprocessing.** In this step, we compute reachability information between each vertex in a bag and vertices appearing in its ancestors in the tree decomposition. Concretely, for each pair of vertices (u, d_1) and (v, d_2) such that u appears in a bag b and v appears in a bag b' that is an ancestor of b, we establish and remember whether $(u, d_1) \rightsquigarrow (v, d_2)$ in $\hat{G}$ and whether $(v, d_2) \rightsquigarrow (u, d_1)$ in $\hat{G}$. As above, we use the notations $(u, d_1) \rightsquigarrow_{\mathsf{anc}} (v, d_2)$ and $(v, d_2) \rightsquigarrow_{\mathsf{anc}} (u, d_1)$.

Steps (1)–(3) above are standard and well-known processes. We now provide details of steps (4)–(6). To skip the details and read about the query phase, see Section 4.3 below.

Step (4): Reduction to Reachability

In this step, our goal is to compute a new graph $\hat{G}$ from the exploded supergraph $\overline{G}$ such that there is a path from (u, d_1) to (v, d_2) in $\hat{G}$ iff there is a same-context valid path from (u, d_1) to (v, d_2) in $\overline{G}$. The idea behind this step is the same as that of the *tabulation algorithm* in [50].

Summary edges. Consider a call vertex c_l in G and its corresponding return-site vertex r_l. For $d_1, d_2 \in D^*$, the edge $((c_l, d_1), (r_l, d_2))$ is called a *summary edge* if there is a same-context valid path from (c_l, d_1) to (r_l, d_2) in the exploded supergraph $\overline{G}$. Intuitively, a summary edge summarizes the effects of procedure calls (same-context interprocedural paths) on the reachability between c_l and r_l. From the definition of *summary edges*, it is straightforward to verify that the graph $\hat{G}$ obtained from $\overline{G}$ by adding every summary edge and removing every interprocedural edge has the desired property, i.e. a pair of vertices are reachable in $\hat{G}$ iff they are reachable by a same-context valid path in $\overline{G}$. Hence, we first find all summary edges and then compute $\hat{G}$. This is shown in Algorithm 1.

We now describe what Algorithm 1 does. Let s_p be the start point of a procedure p. A *shortcut edge* is an edge $((s_p, d_1), (v, d_2))$ such that v is in the same procedure p and there is a same-context valid path from (s_p, d_1) to (v, d_2) in $\overline{G}$. The algorithm creates an empty graph $H = (\overline{V}, E')$. Note that H is implicitly represented by only saving E'. It also creates a queue Q of edges to be added to H (initially $Q = \overline{E}$) and an empty set S which will store the summary edges. The goal is to construct H such that it contains (i) *intraprocedural* edges of $\overline{G}$, (ii) summary edges, and (iii) shortcut edges.

It constructs H one edge at a time. While there is an unprocessed intraprocedural edge $e = ((u, d_1), (v, d_2))$ in Q, it chooses one such e and adds it to H (lines 5–10). Then, if (u, d_1) is reachable from (s_p, d_3) via a same-context valid

Algorithm 1: Computing $\hat{G}$ in Step (4)

```
1  Q ← Ē;
2  S ← ∅;
3  E′ ← ∅;
4  while Q ≠ ∅ do
5  |  Choose e = ((u, d1), (v, d2)) ∈ Q;
6  |  Q ← Q − {e};
7  |  if (u, v) is an interprocedural edge, i.e. a call-to-start or exit-to-return-site
   |   edge then
8  |  |  continue;
9  |  p ← the procedure s.t. u, v ∈ V_p;
10 |  E′ ← E′ ∪ {e};
11 |  foreach d3 s.t. ((s_p, d3), (u, d1)) ∈ E′ do
12 |  |  if ((s_p, d3), (v, d2)) ∉ E′ ∪ Q then
13 |  |  |  Q ← Q ∪ {((s_p, d3), (v, d2))};
14 |  if u = s_p and v = e_p then
15 |  |  foreach (c_l, d3) s.t. ((c_l, d3), (u, d1)) ∈ Ē do
16 |  |  |  foreach d4 s.t. ((v, d2), (r_l, d4)) ∈ Ē do
17 |  |  |  |  if ((c_l, d3), (r_l, d4)) ∉ E′ ∪ Q then
18 |  |  |  |  |  S ← S ∪ {((c_l, d3), (r_l, d4))};
19 |  |  |  |  |  Q ← Q ∪ {((c_l, d3), (r_l, d4))};
20 Ĝ ← Ḡ;
21 foreach e = ((u, d1), (v, d2)) ∈ Ē do
22 |  if u and v are not in the same procedure then
23 |  |  Ĝ = Ĝ − {e};
24 Ĝ ← Ĝ ∪ S;
```

path, then by adding the edge e, the vertex (v, d_2) also becomes accessible from (s_p, d_3). Hence, it adds the shortcut edge $((s_p, d_3), (v, d_2))$ to Q, so that it is later added to the graph H. Moreover, if u is the start s_p of the procedure p and v is its end e_p, then for every call vertex c_l calling the procedure p and its respective return-site r_l, we can add summary edges that summarize the effect of calling p (lines 14–19). Finally, lines 20–24 compute $\hat{G}$ as discussed above.

Correctness. As argued above, every edge that is added to H is either intraprocedural, a summary edge or a shortcut edge. Moreover, all such edges are added to H, because H is constructed one edge at a time and every time an edge e is added to H, all the summary/shortcut edges that might occur as a result of adding e to H are added to the queue Q and hence later to H. Therefore, Algorithm 1 correctly computes summary edges and the graph $\hat{G}$.

Complexity. Note that the graph H has at most $O(|E| \cdot |D^*|^2)$ edges. Addition of each edge corresponds to one iteration of the while loop at line 4 of Algorithm 1. Moreover, each iteration takes $O(|D^*|)$ time, because the loop at line 11 iterates over at most $|D^*|$ possible values for d_3 and the loops at lines 15 and 16 have constantly many iterations due to the bounded bandwidth assump-

tion (Section 2.1). Since $|D^*| = O(|D|)$ and $|E| = O(n)$, the total runtime of Algorithm 1 is $O(|n| \cdot |D|^3)$. For a more detailed analysis, see [50, Appendix].

Step (5): Local Preprocessing

In this step, we compute the set R_{local} of local reachability edges, i.e. edges of the form $((u, d_1), (v, d_2))$ such that u and v appear in the same bag b of a tree decomposition T_i and $(u, d_1) \rightsquigarrow (v, d_2)$ in $\hat{G}$. We write $(u, d_1) \rightsquigarrow_{\mathsf{local}} (v, d_2)$ to denote $((u, d_1), (v, d_2)) \in R_{\mathsf{local}}$. Note that $\hat{G}$ has no interprocedural edges. Hence, we can process each T_i separately. We use a divide-and-conquer technique similar to the kernelization method used in [22] (Algorithm 2).

Algorithm 2 processes each tree decomposition T_i separately. When processing T, it chooses a leaf bag b_l of T and computes all-pairs reachability on the induced subgraph $H_l = \hat{G}[V(b_l) \times D^*]$, consisting of vertices that appear in b_l. Then, for each pair of vertices (u, d_1) and (v, d_2) s.t. u and v appear in b_l and $(u, d_1) \rightsquigarrow (v, d_2)$ in H_l, the algorithm adds the edge $((u, d_1), (v, d_2))$ to both R_{local} and $\hat{G}$ (lines 7–9). Note that this does not change reachability relations in $\hat{G}$, given that the vertices connected by the new edge were reachable by a path before adding it. Then, if b_l is not the only bag in T, the algorithm recursively calls itself over the tree decomposition $T - b_l$, i.e. the tree decomposition obtained by removing b_l (lines 10–11). Finally, it repeats the reachability computation on H_l (lines 12–14). The running time of the algorithm is $O(n \cdot |D^*|^3)$.

Algorithm 2: Local Preprocessing in Step (5)

```
1  R_local ← ∅;
2  foreach T_i do
3  |  computeLocalReachability(T_i);
4  Function computeLocalReachability(T)
5  |  Choose a leaf bag b_l of T;
6  |  b_p ← parent of b_l;
7  |  foreach u, v ∈ V(b_l),  d_1, d_2 ∈ D* s.t. (u, d_1) ⇝ (v, d_2) in Ĝ[V(b_l) × D*]
   |   do
8  |  |  Ĝ = Ĝ ∪ {((u, d_1), (v, d_2))};
9  |  |  R_local = R_local ∪ {((u, d_1), (v, d_2))};
10 |  if b_p ≠ null then
11 |  |  computeLocalReachability(T − b_l);
12 |  |  foreach u, v ∈ V(b_l),  d_1, d_2 ∈ D* s.t. (u, d_1) ⇝ (v, d_2) in
   |  |    Ĝ[V(b_l) × D*] do
13 |  |  |  Ĝ = Ĝ ∪ {((u, d_1), (v, d_2))};
14 |  |  |  R_local = R_local ∪ {((u, d_1), (v, d_2))};
```

Example 7. Consider the graph G and tree decomposition T given in Figure 6 and let $D^* = \{\mathbf{0}\}$, i.e. let $\hat{G}$ and $\bar{G}$ be isomorphic to G. Figure 7 illustrates the

steps taken by Algorithm 2. In each step, a bag is chosen and a local all-pairs reachability computation is performed over the bag. Local reachability edges are added to R_{local} and to $\hat{G}$ (if they are not already in $\hat{G}$).

We now prove the correctness and establish the complexity of Algorithm 2.

Correctness. We prove that when $\mathsf{computeLocalReachability}(T)$ ends, the set R_{local} contains all the local reachability edges between vertices that appear in the same bag in T. The proof is by induction on the size of T. If T consists of a single bag, then the local reachability computation on H_l (lines 7–9) fills R_{local} correctly. Now assume that T has n bags. Let $H_{-l} = \hat{G}[\cup_{b_i \in T, i \neq l} V(b_i) \times D^*]$. Intuitively, H_{-l} is the part of $\hat{G}$ that corresponds to other bags in T, i.e. every bag except the leaf bag b_l. After the local reachability computation at lines 7–9, (v, d_2) is reachable from (u, d_1) in H_{-l} only if it is reachable in $\hat{G}$. This is because (i) the vertices of H_l and H_{-l} form a separation of $\hat{G}$ with separator $(V(b_l) \cap V(b_p)) \times D^*$ (Lemma 3) and (ii) all reachability information in H_l is now replaced by direct edges (line 8). Hence, by induction hypothesis, line 11 finds all the local reachability edges for $T - b_l$ and adds them to both R_{local} and $\hat{G}$. Therefore, after line 11, for every $u, v \in V(b_l)$, we have $(u, d_1) \rightsquigarrow (v, d_2)$ in H_l iff $(u, d_1) \rightsquigarrow (v, d_2)$ in $\hat{G}$. Hence, the final all-pairs reachability computation of lines 12–14 adds all the local edges in b_l to R_{local}.

Complexity. Algorithm 2 performs at most two local all-pair reachability computations over the vertices appearing in each bag, i.e. $O(t \cdot |D^*|)$ vertices. Each such computation can be performed in $O(t^3 \cdot |D^*|^3)$ using standard reachability algorithms. Given that the T_i's have $O(n)$ bags overall, the total runtime of Algorithm 2 is $O(n \cdot t^3 \cdot |D^*|^3) = O(n \cdot |D^*|^3)$. Note that the treewidth t is a constant and hence the factor t^3 can be removed.

Step (6): Ancestors Reachability Preprocessing

This step aims to find reachability relations between each vertex of a bag and vertices that appear in the ancestors of that bag. As in the previous case, we compute a set R_{anc} and write $(u, d_1) \rightsquigarrow_{\mathsf{anc}} (v, d_2)$ if $((u, d_1), (v, d_2)) \in R_{\mathsf{anc}}$.

This step is performed by Algorithm 3. For each bag b and vertex (u, d) such that $u \in V(b)$ and each $0 \leq j < \mathsf{d}_v$, we maintain two sets: $F(u, d, b, j)$ and $F'(u, d, b, j)$ each containing a set of vertices whose first coordinate is in the ancestor of b at depth j. Intuitively, the vertices in $F(u, d, b, j)$ are reachable from (u, d). Conversely, (u, d) is reachable from the vertices in $F'(u, d, b, j)$. At first all F and F' sets are initialized as $\emptyset$. We process each tree decomposition T_i in a top-down manner and does the following actions at each bag:

- If a vertex u appears in both b and its parent b_p, then the reachability data computed for (u, d) at b_p can also be used in b. So, the algorithm copies this data (lines 4–7).
- If $(u, d_1) \rightsquigarrow_{\mathsf{local}} (v, d_2)$, then this reachability relation is saved in F and F' (lines 10–11). Also, any vertex that is reachable from (v, d_2) is reachable from (u, d_1), too. So, the algorithm adds $F(v, d_2, b, j)$ to $F(u, d_1, b, j)$ (line 13). The converse happens to F' (line 14).

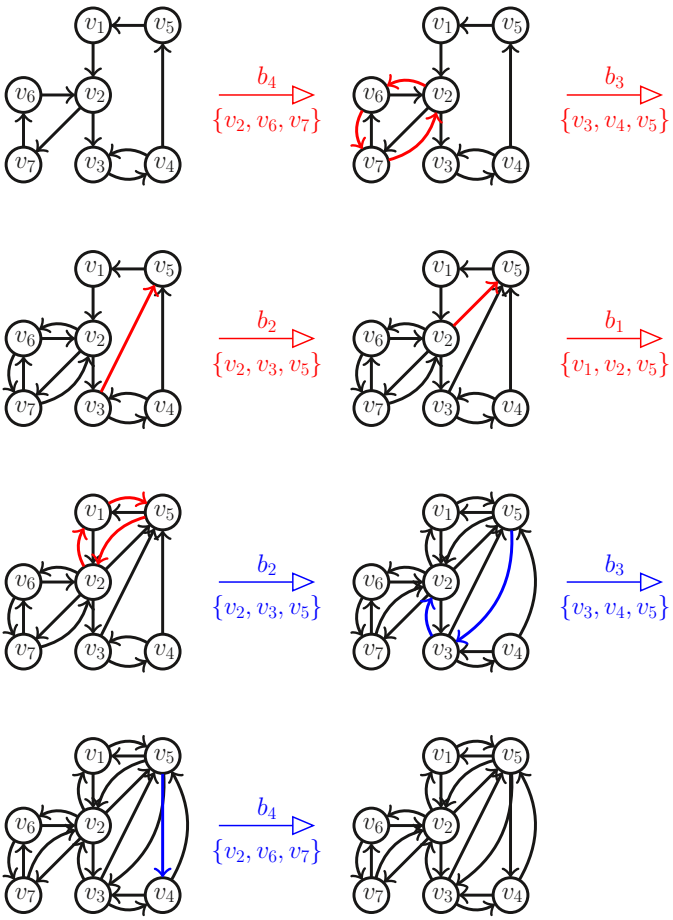

Fig. 7: Local Preprocessing (Step 5) on the graph and decomposition of Figure 6

After the execution of Algorithm 3, we have $(v, d_2) \in F(u, d_1, b, j)$ iff (i) (v, d_2) is reachable from (u, d_1) and (ii) $u \in V(b)$ and $v \in V(\mathsf{a}_b^j)$, i.e. v appears in the ancestor of b at depth j. Conversely, $(u, d_1) \in F'(v, d_2, b, j)$ iff (i) (v, d_2) is reachable from (u, d_1) and (ii) $v \in V(b)$ and $u \in V(\mathsf{a}_b^j)$. Algorithm 3 has a runtime of $O(n \cdot |D|^3 \cdot \log n)$. See [17] for detailed proofs. In the next section, we show that this runtime can be reduced to $O(n \cdot |D|^3)$ using word tricks.

4.2 Word Tricks

We now show how to reduce the time complexity of Algorithm 3 from $O(n \cdot |D^*|^3 \cdot \log n)$ to $O(n \cdot |D^*|^3)$ using word tricks. The idea is to pack the F and F' sets of Algorithm 3 into words, i.e. represent them by a binary sequence.

Algorithm 3: Ancestors Preprocessing in Step (6)

```
1  foreach T_i = (𝔅_i, E_{T_i}) do
2      foreach b ∈ 𝔅_i in top-down order do
3          b_p ← parent of b;
4          foreach u ∈ V(b) ∩ V(b_p), d ∈ D* do
5              foreach 0 ≤ j < d_b do
6                  F(u, d, b, j) ← F(u, d, b_p, j);
7                  F'(u, d, b, j) ← F'(u, d, b_p, j);
8          foreach u, v ∈ V(b), d_1, d_2 ∈ D* do
9              if (u, d_1) ⇝_local (v, d_2) then
10                 F(u, d_1, b, d_b) ← F(u, d_1, b, d_b) ∪ {(v, d_2)};
11                 F'(v, d_2, b, d_b) ← F'(v, d_2, b, d_b) ∪ {(u, d_1)};
12                 foreach 0 ≤ j < d_b do
13                     F(u, d_1, b, j) ← F(u, d_1, b, j) ∪ F(v, d_2, b, j);
14                     F'(v, d_2, b, j) ← F'(v, d_2, b, j) ∪ F'(u, d_1, b, j)
15 R_anc ← {((u, d_1), (v, d_2)) | ∃b, j  (v, d_2) ∈ F(u, d_1, b, j) ∨ (u, d_1) ∈ F'(v, d_2, b, j)};
```

Given a bag b, we define δ_b as the sum of sizes of all ancestors of b. The tree decompositions are balanced, so b has $O(\log n)$ ancestors. Moreover, the width is t, hence $\delta_b = O(t \cdot \log n) = O(\log n)$ for every bag b. We perform a top-down pass of each tree decomposition T_i and compute δ_b for each b.

For every bag b, $u \in V(b)$ and $d_1 \in D^*$, we store $F(u, d_1, b, -)$ as a binary sequence of length $\delta_b \cdot |D^*|$. The first $|V(b)| \cdot |D^*|$ bits of this sequence correspond to $F(u, d_1, b, \mathsf{d}_b)$. The next $|V(b_p)| \cdot |D^*|$ correspond to $F(u, d_1, b, \mathsf{d}_b - 1)$, and so on. We use a similar encoding for F'. Using this encoding, Algorithm 3 can be rewritten by word tricks and bitwise operations as follows:

- Lines 5–6 copy $F(u, d, b_p, -)$ into $F(u, d, b, -)$. However, we have to shift and align the bits, so these lines can be replaced by

$$F(u, d, b, -) \leftarrow F(u, d, b_p, -) \ll |V(b)| \cdot |D^*|;$$

- Line 10 sets a single bit to 1.
- Lines 12–13 perform a union, which can be replaced by the bitwise OR operation. Hence, these lines can be replaced by

$$F(u, d_1, b, -) \leftarrow F(u, d_1, b, -) \textbf{ OR } F(v, d_2, b, -);$$

- Computations on F' can be handled similarly.

Note that we do not need to compute R_{anc} explicitly given that our queries can be written in terms of the F and F' sets. It is easy to verify that using these word tricks, every W operations in lines 6, 7, 13 and 14 are replaced by one or two bitwise operations on words. Hence, the overall runtime of Algorithm 3 is reduced to $O\left(\frac{n \cdot |D^*|^3 \cdot \log n}{W}\right) = O(n \cdot |D^*|^3)$.

4.3 Answering Queries

We now describe how to answer pair and single-source queries using the data saved in the preprocessing phase.

Answering a Pair Query. Our algorithm answers a pair query from a vertex (u, d_1) to a vertex (v, d_2) as follows:

(i) If u and v are not in the same flow graph, return 0 (no).
(ii) Otherwise, let G_i be the flow graph containing both u and v. Let $b_u = \mathsf{rb}(u)$ and $b_v = \mathsf{rb}(v)$ be the root bags of u and v in T_i and let $b = \mathsf{lca}(b_u, b_v)$.
(iii) If there exists a vertex $w \in V(b)$ and $d_3 \in D^*$ such that $(u, d_1) \rightsquigarrow_{\mathsf{anc}} (w, d_3)$ and $(w, d_3) \rightsquigarrow_{\mathsf{anc}} (v, d_2)$, return 1 (yes), otherwise return 0 (no).

Correctness. If there is a path $P : (u, d_1) \rightsquigarrow (v, d_2)$, then we claim P must pass through a vertex (w, d_3) with $w \in V(b)$. If $b = b_u$ or $b = b_v$, the claim is obviously true. Otherwise, consider the path $P' : b_u \rightsquigarrow b_v$ in the tree decomposition T_i. This path passes through b (by definition of b). Let $e = \{b, b'\}$ be an edge of P'. Applying the cut property (Lemma 3) to e, proves that P must pass through a vertex (w, d_3) with $w \in V(b') \cap V(b)$. Moreover, b is an ancestor of both b_u and b_v, hence we have $(u, d_1) \rightsquigarrow_{\mathsf{anc}} (w, d_3)$ and $(w, d_3) \rightsquigarrow_{\mathsf{anc}} (v, d_2)$.

Complexity. Computing LCA takes $O(1)$ time. Checking all possible vertices (w, d_3) takes $O(t \cdot |D^*|) = O(|D|)$. This runtime can be decreased to $O\left(\left\lceil \frac{|D|}{\log n} \right\rceil\right)$ by word tricks.

Answering a Single-source Query. Consider a single-source query from a vertex (u, d_1) with $u \in V_i$. We can answer this query by performing $|V_i| \times |D^*|$ pair queries, i.e. by performing one pair query from (u, d_1) to (v, d_2) for each $v \in V_i$ and $d_2 \in D^*$. Since $|D^*| = O(|D|)$, the total complexity is $O\left(|V_i| \cdot |D| \cdot \left\lceil \frac{|D|}{\log n} \right\rceil\right)$ for answering a single-source query. Using a more involved preprocessing method, we can slightly improve this time to $O\left(\frac{|V_i| \cdot |D|^2}{\log n}\right)$. See [17] for more details. Based on the results above, we now present our main theorem:

Theorem 1. *Given an IFDS instance $I = (G, D, F, M, \cup)$, our algorithm preprocesses I in time $O(n \cdot |D|^3)$ and can then answer each pair query and single-source query in time*

$$O\left(\left\lceil \frac{|D|}{\log n} \right\rceil\right) \quad \textit{and} \quad O\left(\frac{n \cdot |D|^2}{\log n}\right), \quad \textit{respectively.}$$

4.4 Parallelizability and Optimality

We now turn our attention to parallel versions of our query algorithms, as well as cases where the algorithms are optimal.

Parallelizability. Assume we have k threads in our disposal.

1. Given a pair query of the form (u, d_1, v, d_2), let b_u (resp. b_v) be the root bag u (resp. v), and $b = \mathsf{lca}(b_u, b_v)$ the lowest common ancestor of b_u and b_v. We partition the set $V(b) \times D^*$ into k subsets $\{A_i\}_{1 \leq i \leq k}$. Then, thread i handles the set A_i, as follows: for every pair $(w, d_3) \in A_i$, the thread sets the output to 1 (yes) iff $(u, d_1) \rightsquigarrow_{\mathsf{anc}} (w, d_3)$ and $(w, d_3) \rightsquigarrow_{\mathsf{anc}} (v, d_2)$.
2. Recall that a single source query (u, d_1) is answered by breaking it down to $|V_i| \times |D^*|$ pair queries, where G_i is the flow graph containing u. Since all such pair queries are independent, we parallelize them among k threads, and further parallelize each pair query as described above.

With word tricks, parallel pair and single-source queries require $O\left(\left\lceil \frac{|D|}{k \cdot \log n} \right\rceil\right)$ and $O\left(\left\lceil \frac{n \cdot |D|}{k \cdot \log n} \right\rceil\right)$ time, respectively. Hence, for large enough k, each query requires only $O(1)$ time, and we achieve *perfect parallelism*.

Optimality. Observe that when $|D| = O(1)$, i.e. when the domain is small, our algorithm is *optimal*: the preprocessing runs in $O(n)$, which is proportional to the size of the input, and the pair query and single-source query run in times $O(1)$ and $O(n/\log n)$, respectively, each case being proportional to the size of the output. Small domains arise often in practice, e.g. in dead-code elimination or null-pointer analysis.

5 Experimental Results

We report on an experimental evaluation of our techniques and compare their performance to standard alternatives in the literature.

Benchmarks. We used 5 classical data-flow analyses in our experiments, including reachability (for dead-code elimination), possibly-uninitialized variables analysis, simple uninitialized variables analysis, liveness analysis of the variables, and reaching-definitions analysis. We followed the specifications in [36] for modeling the analyses in IFDS. We used real-world Java programs from the DaCapo benchmark suite [6], obtained their flow graphs using Soot [65] and applied the JTDec tool [19] for computing balanced tree decompositions. Given that some of these benchmarks are prohibitively large, we only considered their main Java packages, i.e. packages containing the starting point of the programs. We experimented with a total of 22 benchmarks, which, together with the 5 analyses above, led to a total of 110 instances. Our instance sizes, i.e. number of vertices and edges in the exploded supergraph, range from 22 to 190,591. See [17] for details.

Implementation and comparison. We implemented both variants of our approach, i.e. sequential and parallel, in C++. We also implemented the parts of the classical IFDS algorithm [50] and its on-demand variant [36] responsible for same-context queries. All of our implementations closely follow the pseudocodes of our algorithms and the ones in [50,36], and no additional optimizations are applied. We compared the performance of the following algorithms for randomly-generated queries:

- *SEQ.* The sequential variant of our algorithm.
- *PAR.* A variant of our algorithm in which the queries are answered using perfect parallelization and 12 threads.
- *NOPP.* The classical same-context IFDS algorithm of [50], with *no preprocessing*. NOPP performs a complete run of the classic IFDS algorithm for each query.
- *CPP.* The classical same-context IFDS algorithm of [50], with *complete preprocessing*. In this algorithm, all summary edges and reachability information are precomputed and the queries are simple table lookups.
- *OD.* The on-demand same-context IFDS algorithm of [36]. This algorithm does not preprocess the input. However, it remembers the information obtained in each query and uses it to speed-up the following queries.

For each instance, we randomly generated 10,000 pair queries and 100 single-source queries. In case of single-source queries, source vertices were chosen uniformly at random. For pair queries, we first chose a source vertex uniformly at random, and then chose a target vertex in the same procedure, again uniformly at random.

Experimental setting. The results were obtained on Debian using an Intel Xeon E5-1650 processor (3.2 GHz, 6 cores, 12 threads) with 128GB of RAM. The parallel results used all 12 threads.

Time limit. We enforced a preprocessing time limit of 5 minutes per instance. This is in line with the preprocessing times of state-of-the-art tools on benchmarks of this size, e.g. Soot takes 2-3 minutes to generate all flow graphs for each benchmark.

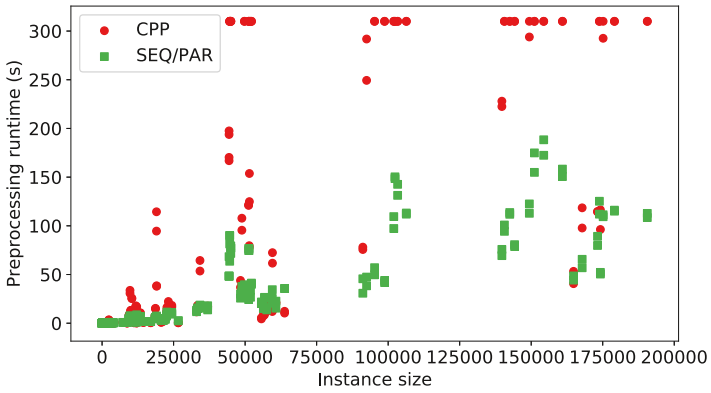

Fig. 8: Preprocessing times of CPP and SEQ/PAR (over all instances). A dot above the 300s line denotes a timeout.

Results. We found that, except for the smallest instances, our algorithm consistently outperforms all previous approaches. Our results were as follows:

Treewidth. The maximum width amongst the obtained tree decompositions was 9, while the minimum was 1. Hence, our experiments confirm the results of [34,19] and show that real-world Java programs have small treewidth. See [17] for more details.

Preprocessing Time. As in Figure 8, our preprocessing is more lightweight and scalable than CPP. Note that CPP preprocessing times out at 25 of the 110 instances, starting with instances of size $< 50,000$, whereas our approach can comfortably handle instances of size $200,000$. Although the theoretical worst-case complexity of CPP preprocessing is $O(n^2 \cdot |D|^3)$, we observed that its runtime over our benchmarks grows more slowly. We believe this is because our benchmark programs generally consist of a large number of small procedures. Hence, the worst-case behavior of CPP preprocessing, which happens on instances with large procedures, is not captured by the DaCapo benchmarks. In contrast, our preprocessing time is $O(n \cdot |D|^3)$ and having small or large procedures does not matter to our algorithms. Hence, we expect that our approach would outperform CPP preprocessing more significantly on instances containing large functions. However, as Figure 8 demonstrates, our approach is faster even on instances with small procedures.

Query Time. As expected, in terms of pair query time, NOPP is the worst performer by a large margin, followed by OD, which is in turn extremely less efficient than CPP, PAR and SEQ (Figure 9, top). This illustrates the underlying trade-off between preprocessing and query-time performance. Note that both CPP and our algorithms (SEQ and PAR), answer each pair query in $O(1)$. They all have pair-query times of less than a millisecond and are indistinguishable in this case. The same trade-off appears in single-source queries as well (Figure 9, bottom). Again, NOPP is the worst performer, followed by OD. SEQ and CPP have very similar runtimes, except that SEQ outperforms CPP in some cases, due to word tricks. However, PAR is extremely faster, which leads to the next point.

Parallelization. In Figure 9 (bottom right), we also observe that single-source queries are handled considerably faster by PAR in comparison with SEQ. Specifically, using 12 threads, the average single-source query time is reduced by a factor of 11.3. Hence, our experimental results achieve near-perfect parallelism and confirm that our algorithm is well-suited for parallel architectures.

Note that Figure 9 combines the results of all five mentioned data-flow analyses. However, the observations above hold independently for every single analysis, as well. See [17] for analysis-specific figures.

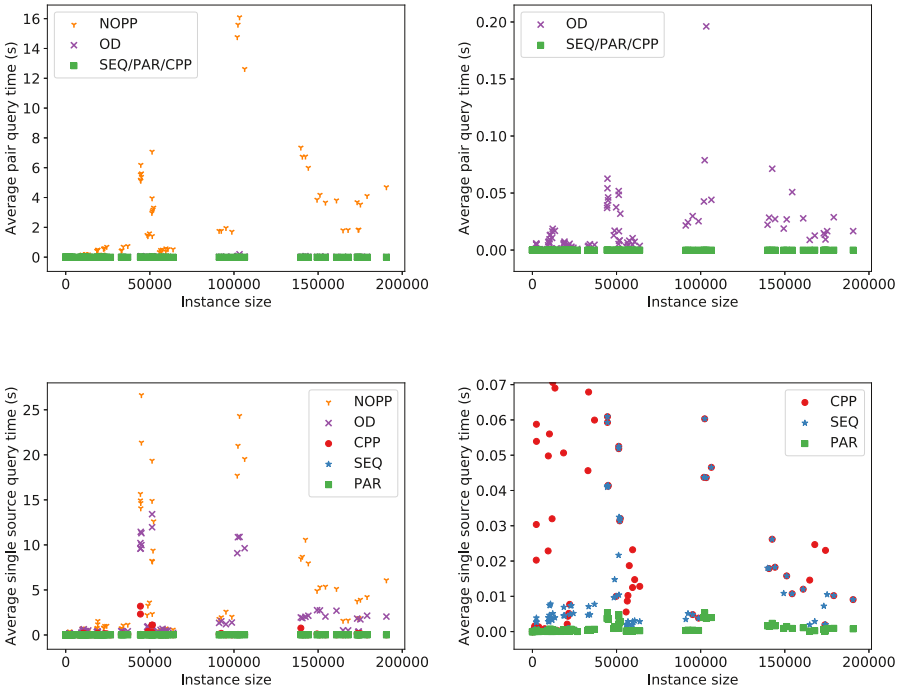

Fig. 9: Comparison of pair query time (top row) and single source query time (bottom row) of the algorithms. Each dot represents one of the 110 instances. Each row starts with a global picture (left) and zooms into smaller time units (right) to differentiate between the algorithms. The plots above contain results over all five analyses. However, our observations hold independently for every single analysis, as well (See [17]).

6 Conclusion

We developed new techniques for on-demand data-flow analyses in IFDS, by exploiting the treewidth of flow graphs. Our complexity analysis shows that our techniques (i) have better worst-case complexity, (ii) offer certain optimality guarantees, and (iii) are embarrassingly paralellizable. Our experiments demonstrate these improvements in practice: after a lightweight one-time preprocessing, queries are answered as fast as the heavyweight complete preprocessing, and the parallel speedup is close to its theoretical optimal. The main limitation of our approach is that it only handles same-context queries. Using treewidth to speedup non-same-context queries is a challenging direction of future work.

References

1. T. J. Watson libraries for analysis (WALA). https://github.com/wala/WALA (2003)
2. Appel, A.W., Palsberg, J.: Modern Compiler Implementation in Java. Cambridge University Press, 2nd edn. (2003)
3. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: PLDI. pp. 259–269 (2014)
4. Babich, W.A., Jazayeri, M.: The method of attributes for data flow analysis. Acta Informatica **10**(3) (1978)
5. Bebenita, M., Brandner, F., Fahndrich, M., Logozzo, F., Schulte, W., Tillmann, N., Venter, H.: Spur: A trace-based JIT compiler for CIL. In: OOPSLA. pp. 708–725 (2010)
6. Blackburn, S.M., Garner, R., Hoffman, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: OOPSLA. pp. 169–190 (2006)
7. Bodden, E.: Inter-procedural data-flow analysis with IFDS/IDE and soot. In: SOAP. pp. 3–8 (2012)
8. Bodden, E., Tolêdo, T., Ribeiro, M., Brabrand, C., Borba, P., Mezini, M.: Spllift: Statically analyzing software product lines in minutes instead of years. In: PLDI. pp. 355–364 (2013)
9. Bodlaender, H., Gustedt, J., Telle, J.A.: Linear-time register allocation for a fixed number of registers. In: SODA (1998)
10. Bodlaender, H.L.: A linear-time algorithm for finding tree-decompositions of small treewidth. SIAM Journal on computing **25**(6), 1305–1317 (1996)
11. Bodlaender, H.L., Hagerup, T.: Parallel algorithms with optimal speedup for bounded treewidth. SIAM Journal on Computing **27**(6), 1725–1746 (1998)
12. Burgstaller, B., Blieberger, J., Scholz, B.: On the tree width of ada programs. In: Ada-Europe. pp. 78–90 (2004)
13. Callahan, D., Cooper, K.D., Kennedy, K., Torczon, L.: Interprocedural constant propagation. In: CC (1986)
14. Chatterjee, K., Choudhary, B., Pavlogiannis, A.: Optimal dyck reachability for data-dependence and alias analysis. In: POPL. pp. 30:1–30:30 (2017)
15. Chatterjee, K., Goharshady, A., Goharshady, E.: The treewidth of smart contracts. In: SAC (2019)
16. Chatterjee, K., Goharshady, A.K., Goyal, P., Ibsen-Jensen, R., Pavlogiannis, A.: Faster algorithms for dynamic algebraic queries in basic RSMs with constant treewidth. ACM Transactions on Programming Languages and Systems **41**(4), 1–46 (2019)
17. Chatterjee, K., Goharshady, A.K., Ibsen-Jensen, R., Pavlogiannis, A.: Optimal and perfectly parallel algorithms for on-demand data-flow analysis. arXiv preprint 2001.11070 (2020)
18. Chatterjee, K., Goharshady, A.K., Okati, N., Pavlogiannis, A.: Efficient parameterized algorithms for data packing. In: POPL. pp. 1–28 (2019)
19. Chatterjee, K., Goharshady, A.K., Pavlogiannis, A.: JTDec: A tool for tree decompositions in soot. In: ATVA. pp. 59–66 (2017)

20. Chatterjee, K., Ibsen-Jensen, R., Goharshady, A.K., Pavlogiannis, A.: Algorithms for algebraic path properties in concurrent systems of constant treewidth components. ACM Transactions on Programming Langauges and Systems **40**(3), 9 (2018)
21. Chatterjee, K., Ibsen-Jensen, R., Pavlogiannis, A.: Optimal reachability and a space-time tradeoff for distance queries in constant-treewidth graphs. In: ESA (2016)
22. Chaudhuri, S., Zaroliagis, C.D.: Shortest paths in digraphs of small treewidth. part i: Sequential algorithms. Algorithmica **27**(3-4), 212–226 (2000)
23. Chaudhuri, S.: Subcubic algorithms for recursive state machines. In: POPL (2008)
24. Chen, T., Lin, J., Dai, X., Hsu, W.C., Yew, P.C.: Data dependence profiling for speculative optimizations. In: CC. pp. 57–72 (2004)
25. Cousot, P., Cousot, R.: Static determination of dynamic properties of recursive procedures. In: IFIP Conference on Formal Description of Programming Concepts (1977)
26. Cygan, M., Fomin, F.V., Kowalik, Ł., Lokshtanov, D., Marx, D., Pilipczuk, M., Pilipczuk, M., Saurabh, S.: Parameterized algorithms, vol. 4 (2015)
27. Duesterwald, E., Gupta, R., Soffa, M.L.: Demand-driven computation of interprocedural data flow. POPL (1995)
28. Dutta, S.: Anatomy of a compiler. Circuit Cellar **121**, 30–35 (2000)
29. Flückiger, O., Scherer, G., Yee, M.H., Goel, A., Ahmed, A., Vitek, J.: Correctness of speculative optimizations with dynamic deoptimization. In: POPL. pp. 49:1–49:28 (2017)
30. Giegerich, R., Möncke, U., Wilhelm, R.: Invariance of approximate semantics with respect to program transformations. In: ECI (1981)
31. Gould, C., Su, Z., Devanbu, P.: Jdbc checker: A static analysis tool for SQL/JDBC applications. In: ICSE. pp. 697–698 (2004)
32. Grove, D., Torczon, L.: Interprocedural constant propagation: A study of jump function implementation. In: PLDI (1993)
33. Guarnieri, S., Pistoia, M., Tripp, O., Dolby, J., Teilhet, S., Berg, R.: Saving the world wide web from vulnerable javascript. In: ISSTA. pp. 177–187 (2011)
34. Gustedt, J., Mæhle, O.A., Telle, J.A.: The treewidth of java programs. In: ALENEX. pp. 86–97 (2002)
35. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM Journal on Computing **13**(2), 338–355 (1984)
36. Horwitz, S., Reps, T., Sagiv, M.: Demand interprocedural dataflow analysis. ACM SIGSOFT Software Engineering Notes (1995)
37. Hovemeyer, D., Pugh, W.: Finding bugs is easy. ACM SIGPLAN Notices **39**(12), 92–106 (Dec 2004)
38. Klaus Krause, P., Larisch, L., Salfelder, F.: The tree-width of C. Discrete Applied Mathematics (03 2019)
39. Knoop, J., Steffen, B.: The interprocedural coincidence theorem. In: CC (1992)
40. Krüger, S., Späth, J., Ali, K., Bodden, E., Mezini, M.: CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. In: ECOOP. pp. 10:1–10:27 (2018)
41. Lee, Y.f., Marlowe, T.J., Ryder, B.G.: Performing data flow analysis in parallel. In: ACM/IEEE Supercomputing. pp. 942–951 (1990)
42. Lee, Y.F., Ryder, B.G.: A comprehensive approach to parallel data flow analysis. In: ICS. pp. 236–247 (1992)

43. Lin, J., Chen, T., Hsu, W.C., Yew, P.C., Ju, R.D.C., Ngai, T.F., Chan, S.: A compiler framework for speculative optimizations. ACM Transactions on Architecture and Code Optimization **1**(3), 247–271 (2004)
44. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann (1997)
45. Naeem, N.A., Lhoták, O., Rodriguez, J.: Practical extensions to the ifds algorithm. CC (2010)
46. Nanda, M.G., Sinha, S.: Accurate interprocedural null-dereference analysis for java. In: ICSE. pp. 133–143 (2009)
47. Rapoport, M., Lhoták, O., Tip, F.: Precise data flow analysis in the presence of correlated method calls. In: SAS. pp. 54–71 (2015)
48. Reps, T.: Program analysis via graph reachability. ILPS (1997)
49. Reps, T.: Undecidability of context-sensitive data-dependence analysis. ACM Transactions on Programming Languages and Systems **22**(1), 162–186 (2000)
50. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: POPL. pp. 49–61 (1995)
51. Reps, T.: Demand interprocedural program analysis using logic databases. In: Applications of Logic Databases, vol. 296 (1995)
52. Robertson, N., Seymour, P.D.: Graph minors. iii. planar tree-width. Journal of Combinatorial Theory, Series B **36**(1), 49–64 (1984)
53. Rodriguez, J., Lhoták, O.: Actor-based parallel dataflow analysis. In: CC. pp. 179–197 (2011)
54. Rountev, A., Kagan, S., Marlowe, T.: Interprocedural dataflow analysis in the presence of large libraries. In: CC. pp. 2–16 (2006)
55. Sagiv, M., Reps, T., Horwitz, S.: Precise interprocedural dataflow analysis with applications to constant propagation. Theoretical Computer Science (1996)
56. Schubert, P.D., Hermann, B., Bodden, E.: PhASAR: An inter-procedural static analysis framework for C/C++. In: TACAS. pp. 393–410 (2019)
57. Shang, L., Xie, X., Xue, J.: On-demand dynamic summary-based points-to analysis. In: CGO. pp. 264–274 (2012)
58. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Program flow analysis: Theory and applications. Prentice-Hall (1981)
59. Smaragdakis, Y., Bravenboer, M., Lhoták, O.: Pick your contexts well: Understanding object-sensitivity. In: POPL. pp. 17–30 (2011)
60. Späth, J., Ali, K., Bodden, E.: Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. In: POPL. pp. 48:1–48:29 (2019)
61. Sridharan, M., Bodík, R.: Refinement-based context-sensitive points-to analysis for java. ACM SIGPLAN Notices **41**(6), 387–400 (2006)
62. Sridharan, M., Gopan, D., Shan, L., Bodík, R.: Demand-driven points-to analysis for java. In: OOPSLA. pp. 59–76 (2005)
63. Thorup, M.: All structured programs have small tree width and good register allocation. Information and Computation **142**(2), 159–181 (1998)
64. Torczon, L., Cooper, K.: Engineering a Compiler. Morgan Kaufmann, 2nd edn. (2011)
65. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L.J., Lam, P., Sundaresan, V.: Soot - a Java bytecode optimization framework. In: CASCON. p. 13 (1999)
66. Xu, G., Rountev, A., Sridharan, M.: Scaling cfl-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In: ECOOP (2009)
67. Yan, D., Xu, G., Rountev, A.: Demand-driven context-sensitive alias analysis for java. In: ISSTA. pp. 155–165 (2011)

68. Yuan, X., Gupta, R., Melhem, R.: Demand-driven data flow analysis for communication optimization. Parallel Processing Letters **07**(04), 359–370 (1997)
69. Zheng, X., Rugina, R.: Demand-driven alias analysis for c. In: POPL. pp. 197–208 (2008)