GraCFL: A Holistically Designed Vertex-Centric Graph System for CFL Reachability

Sakib Fuad
University of California, Riverside
Riverside, CA, USA
sfuad001@ucr.edu

Amir Hossein Nodehi Sabet
University of California, Riverside
Riverside, CA, USA
anode001@ucr.edu

Umar Farooq
University of California, Riverside
Riverside, CA, USA
ufaro001@ucr.edu

Zhijia Zhao
University of California, Riverside
Riverside, CA, USA
zhijia@cs.ucr.edu

Abstract

Many program analyses can be formulated as context-free language (CFL) reachability problems on an edge-labeled graph. While graph systems have been proposed recently for large-scale CFL reachability analysis of system software, the design space has not yet been systematically explored, leading to sub-optimal performance.

This work presents *GraCFL* [1], a holistically designed graph system for CFL reachability. Inspired by the vertex-centric processing paradigm, we formalize CFL reachability using a *multi-directional vertex-centric* model. We then analyze this model in terms of computation redundancy, strategies for deriving new reachability, data locality, and parallelism. The analysis reveals a set of insights that guide the design of new techniques and optimizations to improve system performance. As a result of the systematic design, *GraCFL* demonstrates superior performance compared to state-of-the-art graph systems, with an average 14.14× speedup over *Graspan* and 8.33× speedup over *POCR*. Its source code is available at https://github.com/AutomataLab/GraCFL.

CCS Concepts

• **Theory of computation** → **Grammars and context-free languages**; **Graph algorithms analysis**.

Keywords

Context-Free Language, CFL Reachability, Graph System, Vertex-centric Model

ACM Reference Format:
Sakib Fuad, Amir Hossein Nodehi Sabet, Umar Farooq, and Zhijia Zhao. 2025. GraCFL: A Holistically Designed Vertex-Centric Graph System for CFL Reachability. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3721145.3725762

1 Introduction

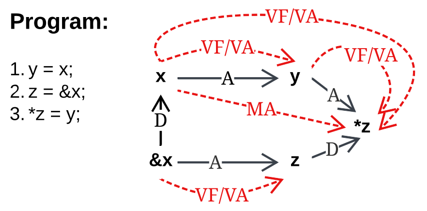
Context-free language (CFL) reachability is a fundamental problem in program analysis [9, 54, 81], with many important applications, including pointer analysis [49, 61, 66–68, 75, 82], interprocedural data flow analysis [2, 34, 53, 54, 74], program slicing [55], type-based flow analysis [46, 50, 51], and shape analysis [52]. Solving these analyses is essential for various optimizations, debugging, and security measures [73].

Problem Definition. Consider a context-free language L over alphabet Σ, and a graph (V, E), where edges are labeled using symbols from Σ. A vertex v is L-*reachable* from vertex s if there exists a path from s to v where the concatenation of edge labels form a string in L.

Figure 1 illustrates the alias analysis for C programs using CFL reachability [82], which is employed by LLVM [40, 42] to support optimizations like loop invariant code motion, dead code elimination, global value numbering and more [41]. First, an initial edge-labeled graph is generated based on the input program (black edges in Figure 1a), where edges encode the "assignment" (A) and "dereference" (D) relations. Meanwhile, the alias analysis is specified in a context-free grammar (CFG), as shown in Figure 1b. For example, the *VF* rule indicates that an assignment followed by a memory alias (optional) or any repetition of this pattern is a valid value flow. Considering the A-edge from vertex x to y in the

[1] GraCFL is pronounced as "graceful".

This work is licensed under a Creative Commons Attribution 4.0 International License.
ICS '25, Salt Lake City, UT, USA
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3725762



(a) Alias analysis example.

Value flow:	VF	$::=$	$(A \;\; MA?)^*$
Memory alias:	MA	$::=$	$\overline{D} \;\; VA \;\; D$
Value alias:	VA	$::=$	$\overline{VF} \;\; MA? \;\; VF$

($\overline{D}$ and $\overline{VF}$ indicate edges in reversed direction.)

(b) Context-free grammar

Figure 1: Alias Analysis via CFL Reachability

initial graph, *VF* rule implies that y is *VF*-reachable from x. Similarly, according to the other two rules, it is possible to infer the *MA*-reachability and *VA*-reachability, which capture the memory aliases and value aliases, respectively.

For real-world large system software, such as the Linux kernel and the PostgreSQL database, the initial graph for interprocedural alias analysis can be large, often containing millions or even tens of millions of vertices and edges [73]. It is challenging to perform CFL reachability analysis at this scale due to its resource-demanding nature—it takes at least subcubic time [6] to find the reachability between every pair of vertices. Moreover, the analysis generates a large number of new edges that are dynamically inserted into the graph.

State of The Art. Two recently developed graph systems for large-scale CFL reachability analysis are *Graspan* [73, 85] and *POCR* [30]. The latter is built on SVF [69]—a popular static analysis framework. They take as input an initial graph and a normalized CFG, where the graph encodes immediate facts of a program, while the CFG captures the analysis logic. Then, by iteratively examining the graph according to the CFG, these systems gradually derive new edges that represent newly discovered reachability and insert them back into the graph, until no new edges can be identified. Figure 1a shows the final graph with new edges, denoted as dashed red arrows, that represent the aliasing relations. For example, the labeled edges $(x, *z, VA)$ and $(x, *z, MA)$ indicate that x and $*z$ are both value aliases and memory aliases.

While serving the same purpose, the two graph systems follow completely different designs under the hood. Starting from the underlying computation model, *Graspan* initiates computations per vertex—a vertex-centric approach, while *POCR* follows the classic worklist algorithm [45], defining computations on edges—an edge-centric design. To avoid

Table 1: GraCFL vs. Graspan and POCR

	Graspan [73]	**POCR** [30]	**GraCFL** (this work)
Model	v-centric (FW)	edge-centric	v-centric (FW/BW/BI)
Redun.	local lists	global worklist	local temporal vector
Derivation	topo-driven	grammar-driven	topo/grammar-driven
Locality	low	high	tunable by direction
Para.	high	low	
Edges	sorted vectors/arrays	sparse bit vector +spanning trees	vectors+hashsets

repeatedly examining the same edges across iterations, *POCR* maintains a global worklist containing only newly derived edges and processes only cases involving the new edges. In contrast, *Graspan* keeps new edges local to each vertex and merge them into the "old" edge list in each iteration. Finally, to avoid adding duplicate edges, *Graspan* sorts edges for fast de-duplication, while *POCR* uses sparse bit vectors to keep edges unique. Table 1 summarizes these design differences, along with other differences that will be discussed later.

Beyond the above design aspects, *Graspan* employs a novel out-of-core processing scheme that partitions the graph and loads pairs of partitions for processing. In contrast, *POCR* operates in memory with unique algorithmic optimizations by exploiting properties of recursive grammars.

Contributions. As discussed above, the design space for a graph system supporting CFL reachability is complex. The primary goal of this work is to systematically explore this design space, with a focus on the vertex-centric design of an in-memory graph system for general context-free grammars. Specifically, it makes the following series of contributions.

- First, it systematically defines the *vertex-centric model* for CFL reachability computations and highlights a key design aspect, *model direction*, which could be either *forward*, *backward*, or *bidirectional*.
- Second, it proposes a data structure called *temporal vector*, which uses a pair of pointers to separate edges generated at different times, enabling fast merging of new edges by simply sliding the pointers.
- Third, it characterizes edge derivation strategies as *topology-driven* and *grammar-driven* approaches, and selects the appropriate one based on the inputs.
- Fourth, it identifies a key data locality optimization for the bidirectional model and uncovers discrepancies in both locality and parallelism across different model directions, highlighting the need of direction selection.
- Finally, it proposes to maintain the growing graph using both vectors and hashsets, enabling efficient edge list traversal and fast edge existence checks, at the cost of increased memory usage.

Based on the above exploration, we developed *GraCFL*, a versatile graph system that offers customizable execution models that can be configured according to users' specific constraints, such as the available CPU cores and memory capacities. We compared *GraCFL* with *Graspan-C* [85] and *POCR* [30], two state-of-the-art graph systems for solving CFL reachability analysis, using two points-to analyses and one data-flow analysis on a suite of system software, which include Linux, PostgreSQL, Apache httpd, Hadoop HDFS, and MapReduce. *GraCFL* exhibits significant speedups over the existing systems, achieving on average 14.14× and 8.33× speedups over *Graspan-C* and *POCR*, respectively.

2 Background

This section formally defines the CFL reachability problem and presents the basic workflow for solving it.

2.1 Problem Formalization

Consider an edge-labeled graph $G(V, E, \Sigma)$, where the edge labels are symbols from the alphabet Σ, and the set of edges is denoted as $E = \{(v_i, v_j, a) | a \in \Sigma \text{ and } v_i, v_j \in V\}$. A path π on the graph is a sequence of consecutive edges: $\pi = \{(v_1, v_2, a_1), (v_2, v_3, a_2), \ldots, (v_{k-1}, v_k, a_{k-1})\}$ where $k-1$ is the length of the path. A path string $l(\pi)$ is defined as a serial concatenation of the edge labels over the path π:

$$l(\pi) = (a_1, a_2, \ldots, a_{k-1}) \tag{1}$$

Consider a context-free grammar $\mathcal{G}$ $(\mathcal{V}, \Sigma, R, S)$, where $\mathcal{V}$ and Σ are two disjoint sets, for non-terminals and terminals, respectively, R is the set of production rules, and S denotes the start symbol (a non-terminal). A production rule maps a non-terminal to a string with symbols from $\mathcal{V} \cup \Sigma$, that is, $R = \{A ::= \alpha | A \in \mathcal{V} \text{ and } \alpha \text{ is a string in } (\mathcal{V} \cup \Sigma)^*\}$ where asterisk $*$ denotes the Kleene star operation. The context-free language defined by grammar $\mathcal{G}$ is denoted as $L(\mathcal{G})$.

Definition 2.1. Given a context-free grammar $\mathcal{G}(\mathcal{V}, \Sigma, R, S)$ and a graph $G(V, E, \Sigma)$, where the labels in the graph share the same symbols as the terminal set of the grammar—Σ. If there exists at least one path π from vertex s to vertex t $(s, t \in V)$ such that the corresponding path string $l(\pi)$ can be derived from non-terminal A according to grammar $\mathcal{G}$, then t is A-reachable from s. If A is the start non-terminal, we say t is S-reachable or L-reachable from s, where L is the language defined by grammar $\mathcal{G}$.

2.2 Graph Systems for CFL Reachability

The existing graph systems, like *Graspan* [73] and *POCR* [30], follow a similar high-level processing workflow. They take as input an initial graph $G(V, E)$ and a context-free grammar $\mathcal{G}$, and output a new graph $G'(V', E')$ where $V' = V$ and $E' \supseteq E$. The new edges in G' (i.e., $E' \setminus E$) capture the A-reachability, where A is a non-terminal (i.e., $A \in \mathcal{V}$).

CFG Normalization. To simplify system design, these graph systems assume a normal grammar [2], where the right-hand side (RHS) of each production rule has at most two symbols. For example, rule $A ::= AbB$ could be normalized to

$$A ::= AA' \text{ and } A' ::= bB.$$

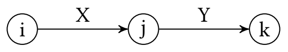
This conversion takes linear time, and the grammar size also increases linearly. After the normalization, each production must be in one of the following three forms

$$A ::= XY \tag{2}$$

$$A ::= X \tag{3}$$

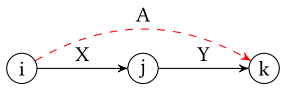
$$A ::= \epsilon \tag{4}$$

New Edge Derivation. The graph systems derive new edges according to the normalized grammar. Thanks to the normalization, the systems only need to check at most two adjacent edges each time. Consider the following two edges.

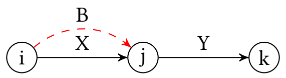


The systems need to cover the following three cases.

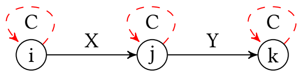
- Check if string XY matches the RHS of any grammar rule. For example, if there exists a rule like $A ::= XY$, then an edge (v_i, v_k, A) is generated;



- Check if individual symbols X and Y match the RHS of any grammar rule. For example, if there exists a rule like $B ::= X$, then an edge (v_i, v_j, B) is generated;



- Check if there exists rules like $C ::= \epsilon$, if so, generate a self-loop edge for each vertex. In this case, three edges (v_i, v_i, C), (v_j, v_j, C), and (v_k, v_k, C) are generated.



Note that the last case depends only on vertices, thus can be preprocessed. In the following, we will focus our discussion on the first two cases.

Fixed-Point Iterative Algorithm. Since the newly generated edges can be used to generate more edges, these systems must iteratively derive and add new edges until no more can be generated—reaching a fixed point. To achieve this, *Graspan* examines all the vertices in each iteration until no

[2]This normal form is similar to Chomsky Normal Form, but it allows ϵ-rules and doesn't limit where terminal symbols can appear.

Algorithm 1 Vertex-Centric Model

```
1: $continue$ = TRUE /* a global flag for termination */
2: while $continue$ do /* fixed-point iterations */
3:     $continue$ = FALSE
4:     for vertex $v_i \in V$ do
5:         $f(v_i)$ /* vertex function may set $continue$ = TRUE */
```

vertices consist of new edges. In comparison, *POCR* keeps a worklist of new edges alongside the graph, continuously appending new edges to the worklist and removing edges after being processed, until the worklist becomes empty. The worst-case time complexity for solving the general CFL reachability problem has been proven to be subcubic [6] to the number of vertices and the size of the alphabet.

Graph Representation. To represent the growing graph, systems need to use dynamic data structures, like adjacency lists. *Graspan* uses vectors to store edge lists, while *POCR* uses hashsets along with spanning trees. As we will discuss later, each choice has its own advantages and disadvantages in the context of CFL reachability analysis.

In addition, both *Graspan* and *POCR* provide some unique features. *Graspan* supports scenarios with limited memory budgets using out-of-core processing. This is achieved with an adaptive partitioning-scheduling scheme. A key feature of *POCR* is its redundancy elimination for recursive grammars, which leverages edge derivation order to avoid the repetitive derivation of identical edges. In contrast, this work targets *system-level* design and optimizations of an *in-memory* graph system for *general* context-free grammars.

3 Computation Model

Vertex-centric model [43, 44] is an established paradigm for solving many classic iterative graph problems, such as breadth-first search (BFS), shortest path, and PageRank [4]. As described in Algorithm 1, under this model, computations are defined from the perspective of a vertex, expressed by a function called the *vertex function* $f(v)$.

Taking single-source shortest path (SSSP) problem as an example, the vertex function uses v's latest shortest distance (from the source) to update the shortest distance of each of v's out-neighbors (from the source), that is,

$$dist_n = \min\{dist_n, dist_v + weight(v, n)\} \quad (5)$$

for each $n \in outnbrs(v)$. By applying this function to all vertices (or a subset of vertices) in the graph iteration by iteration, the shortest distances from the source to all the other vertices in the graph can be obtained.

Vertex-Centric CFL Reachability. Despite the significant differences in the computations (min/max vs. CFG matching), we find that the CFL reachability problem naturally suits

Algorithm 2 Forward Vertex Function

```
1: function $f_{FW}(v_i)$
2:     for edge $(v_i, v_j, B)$ in $OE(v_i)$ do
3:         for production $A ::= B \in \mathcal{G}$ do
4:             if edge $(v_i, v_j, A) \notin OE(v_i)$ then
5:                 add $(v_i, v_j, A)$ into $OE(v_i)$
6:                 $continue$ = TRUE
7:         for edge $(v_j, v_k, C)$ of $OE(v_j)$ do
8:             for production $A ::= BC \in \mathcal{G}$ do
9:                 if edge $(v_i, v_k, A) \notin OE(v_i)$ then
10:                    add $(v_i, v_k, A)$ into $OE(v_i)$
11:                    $continue$ = TRUE
```

the vertex-centric model. An intuitive way to define the vertex function, which was first introduced by *Graspan* [73], is illustrated in Figure 2a. Given a vertex v_i, the function scans its out-edges $OE(v_i)$, and for each out-edge (v_i, v_j, C), it further traverses v_j's out-edges $OE(v_j)$. By concatenating the labels on two adjacent edges and checking them against the grammar, it determines if a new edge is derived for vertex v_i. Algorithm 2 outlines the above processing logic.

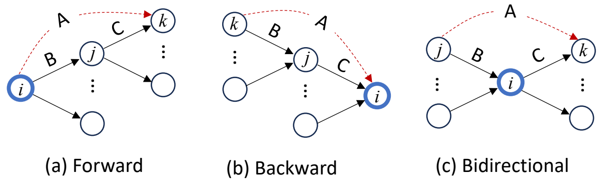


Figure 2: Directions of Vertex-Centric Model

Directions. In fact, Algorithm 2 only shows one possible vertex function, which we call the *forward vertex function*. Instead of scanning the one-hop and two-hop out-neighbors, the vertex function could traverse its one-hop and two-hop in-neighbors (see Figure 2b), referred to as the *backward vertex function*. This distinction in traversal direction mirrors the push/pull model [3, 15, 77] in general-purpose graph systems, which has been proven to be crucial to performance.

In fact, there is the third way to define the vertex function: traverse the vertex's direct in-neighbors and out-neighbors, concatenate their labels on each combination of two adjacent edges, as depicted by Figure 2c, and detailed by Algorithm 3. We refer to this approach as the *bidirectional vertex function*, which resembles the gather-apply-scatter (GAS) model [13] used in some general-purpose graph systems.

Table 2 compares the vertex-centric model for classic graph problems (like BFS and shortest paths) and that for CFL reachability.

[3]The graph is fixed during query evaluation to ensure result integrity.

Algorithm 3 Bidirectional Vertex Function

```
function $f_{BI}(v_i)$
    for edge $(v_j, v_i, B)$ in $IE(v_i)$ do
        for production $A ::= B \in \mathcal{G}$ do
            if edge $(v_i, v_j, A) \notin IE(v_i)$ then
                add $(v_i, v_j, A)$ into $IE(v_i)$
                add $(v_i, v_j, A)$ into $OE(v_j)$
                $continue$ = TRUE
        for edge $(v_i, v_k, C)$ in $OE(v_i)$ do
            for production $A ::= BC \in \mathcal{G}$ do
                if edge $(v_j, v_k, A) \notin IE(v_k)$ then
                    add $(v_j, v_k, A)$ into $IE(v_k)$
                    add $(v_j, v_k, A)$ into $OE(v_j)$
                    $continue$ = TRUE
```

Table 2: Comparison of Vertex-Centric Models

	Classic Graph Problems	CFL Reachability
Edge Set	static (fixed)[3]	dynamic (growing)
Traversal	one-hop neighbors	1-2 hops of neighbors
Comp.	min/max	CFG matching
Direction	push/pull/GAS	FW/BW/BI

Graph Representations. The forward and backward models require an out-neighbor adjacency list and an in-neighbor adjacency list, respectively. In comparison, the bidirectional model needs both the out- and in-neighbor adjacency lists, which doubles the memory cost. It is worth noting that the classic worklist-based algorithm for CFL reachability [45] also requires both out- and in-neighbor adjacency lists.

In addition, it is important to note that when new edges are inserted, the forward and backward models add them to the vertex being processed—v_i, while bidirectional model inserts each new edge twice—one into the out-neighbor list of v_i's in-neighbor, and the other into the in-neighbor list of v_i's out-neighbor (see Algorithm 3).

4 Redundancy Elimination

Eliminating redundant computations has been a primary focus of prior graph systems [30, 63, 73]. In this section, we examine the redundancies in vertex-centric models—*repeated edge checks* and present a key finding—the existing approach for avoiding repeated edge checks is costly due to frequent vector copy operations. To avoid vector copies, we propose an alternative solution called *sliding pointers.*

Redundancy. Under a naive design, a graph system may unnecessarily re-examine the same edge or edge pairs across different iterations. As demonstrated in Figure 3, in the I-th iteration, the system checks edge pair BC and subsequently creates a new edge (v_i, v_l, A). In the next iteration, if the system checks the same edge pair BC again, it would be

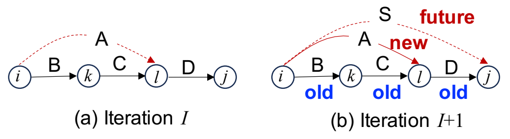


Figure 3: Demonstration of Redundant Checking

Algorithm 4 $f_{FW}()$ with Redundancy Elimination

```
function $f_{FW}(v_i)$
    for edge $(v_i, v_j, B)$ in $newOE(v_i)$ do
        for production $A ::= B \in \mathcal{G}$ do
            if edge $(v_i, v_j, A) \notin OE(v_i)$ then
                add edge $(v_i, v_j, A)$ to $futureOE(v_i)$
                $continue$ = TRUE
        for edge $(v_j, v_k, C)$ in $oldOE(v_j) \cup newOE(v_j)$ do
            if production $A ::= BC \in \mathcal{G}$ then
                if edge $(v_i, v_k, A) \notin OE(v_i)$ then
                    add edge $(v_i, v_k, A)$ to $futureOE(v_i)$
                    $continue$ = TRUE
    for edge $(v_i, v_j, B)$ in $oldOE(v_i)$ do
        for edge $(v_j, v_k, C)$ in $newOE(v_j)$ do
            if production $A ::= BC \in \mathcal{G}$ then
                if edge $(v_i, v_k, A) \notin OE(v_i)$ then
                    add edge $(v_i, v_k, A)$ to $futureOE(v_i)$
                    $continue$ = TRUE
```

a redundant check. Instead, the system should check edge pair AD because edge (v_i, v_l, A) was just created in the prior iteration and hasn't been examined up to this point.

Based on the above observation, the graph system should only check edges *newly* generated in the prior iteration and edge pairs with at least one of those newly generated edges. To distinguish such cases from the rest, *Graspan* categorizes edges into three groups [73]:

- E_{old}: old edges generated before the prior iteration;
- E_{new}: new edges generated in the prior iteration;
- E_{future}: future edges generated in the current iteration.

Then, the system only needs to examine "new" edges and edge pairs in "old-new", "new-old", and "new-new" patterns, as illustrated in Algorithm 4. For the first iteration, all initial edges are marked "new". At the end of each iteration, the graph system updates the edge groups (see Algorithm 5).

Intuitive Implementation. To adopt the "old-new-future" optimization, it is intuitive to maintain three edge vectors for each vertex, one for each type of edges, as illustrated in Figure 4a. In-between two iterations, move new edges from E_{new} to E_{old} and turn the future edge list E_{future} into the new edge list E_{new}. However, the cost of moving edges is not cheap. Essentially, it loops over all the elements in the source

Algorithm 5 Vertex-Centric Model with Redun. Elim.

```
1: continue = TRUE /* a global flag for termination */
2: while continue do /* fixed-point iterations */
3:     continue = FALSE
4:     for vertex $v_i$ in $V$ do
5:         $f(v_i)$ /* vertex function */
6:     updateOldNewFuture() /* add $newOE(v_i)$ into $oldOE(v_i)$,
       move $futureOE(v_i)$ to $newOE(v_i)$, and empty $futureOE(v_i)$ */
```

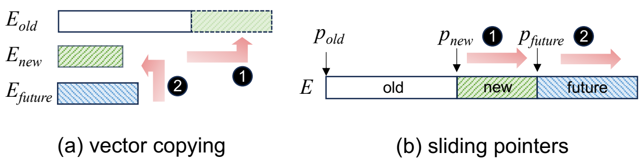


Figure 4: Vector Copying vs. Sliding Pointers

list and move each of them to the destination list. Moreover, the edge movements occur per vertex and per iteration.

Sliding Pointers. To address the issue, we propose a new implementation based on the following observation:

> ***Observation***: *Each generated edge follows a strict chronological "lifecycle": it begins as a "future" edge, then transitions to a "new" edge, and ultimately stays as an "old" edge. No edges are deleted during this process.*

This observation implies the possibility of using a *single vector* to store all the edges of a vertex in the order they are created. To track their status, we can use two pointers, p_{new} and p_{future}, to split the vector into three time zones, as illustrated in Figure 4b. To update their statuses, we simply "slide" the new-edge and future-edge pointers towards the future end. We refer to this data structure as the *temporal vector* and the corresponding technique as *sliding pointers*.

5 Edge Derivation Strategies

Topology-driven Edge Derivation. So far, our discussion has followed an intuitive process for deriving edges, as illustrated in Figure 5: first, locate existing edge(s) and their label(s) (e.g., BC); then, search for matching grammar rules. For each matched rule (e.g., $A ::= BC$), generate a corresponding new edge (an A-edge). We refer to this strategy as *topology-driven edge derivation* , which was employed in *Graspan* [73].

However, there is a caveat in the above strategy—when the graph consists of a diverse set of edge symbols, that is, the size of alphabet Σ is relatively large, it is likely many edge pairs may fail to match any grammar rules. In such cases, no new edge would be generated—these edge pair checks become "wasted". To address this type of inefficiency, we next discuss an alternative strategy for edge derivation.

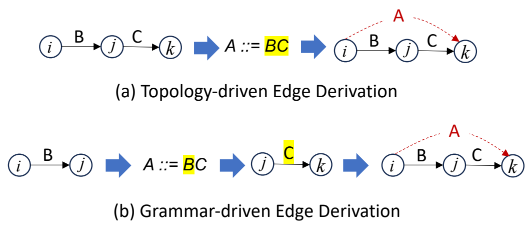

i
B
j
C
k
A ::= BC
A
(a) Topology-driven Edge Derivation
A ::= BC
C
(b) Grammar-driven Edge Derivation

Figure 5: Edge Derivation Strategies

Grammar-driven Edge Derivation. In this design, the graph system leverages grammar rules to guide the derivation of new edges, rather than relying solely on the graph structure. Figure 5b illustrates this alternative strategy. It starts from a single edge (v_i, v_j, B), then searches for grammar rules like $A ::= BC$, where C dictates the label of the second edge. In this way, it only needs to locate the out-edges of v_j with label C rather than scanning all its out-edges. For each matched out-edge of (v_j, v_k, C), it generates a new edge (v_i, v_k, A). Algorithm 6 implements this in the forward model. Note that to quickly locate edges of a particular label, it would be better to index the edges by labels (see Section 7.2), which may further scatter the edges in the adjacency list, potentially worsening the locality.

This grammar-driven strategy was first introduced in the worklist-based solver for CFL reachability [45]. We adopted it to the vertex-centric model. In practice, we found that the grammar-driven strategy fail to outperform the topology-driven strategy for simple grammars (see Section 7), likely due to the limited benefits and the use of label-indexed graph. Therefore, the edge derivation strategy should be selected based on the grammar complexity.

6 Locality, Parallelism, and Trade-offs

Data locality and parallelism are two other critical aspects of graph systems that significantly impact performance. This section examines them within the proposed vertex-centric models, assuming the optimizations discussed in Section 4.

6.1 Data Locality

After measuring the performance of the vertex-centric model under different directions (see data in Section 8), we make two interesting observations regarding its data locality.

First, we find that there exists a discrepancy between the forward and backward models in terms of data locality.

> ***Forward vs. Backward***: *Despite the symmetry between the forward and backward vertex-centric models, their data locality can vary substantially.*

Algorithm 6 f_{FW} with Grammar-driven Edge Derivation

```
1:  function $f_{FW}(v_i)$
2:      for edge $(v_i, v_j, B)$ in $newOE(v_i)$ do
3:          for production $A ::= B \in \mathcal{G}$ do
4:              if edge $(v_i, v_j, A) \notin OE(v_i)$ then
5:                  add edge $(v_i, v_j, A)$ to $futureOE(v_i)$
6:                  $continue$ = TRUE
7:          for production $A ::= BC \in \mathcal{G}$ do
8:              for edge $(v_j, v_k, C)$ in $oldOE(v_j) \cup newOE(v_j)$ do
9:                  if edge $(v_i, v_k, A) \notin OE(v_i)$ then
10:                     add edge $(v_i, v_k, A)$ to $futureOE(v_i)$
11:                     $continue$ = TRUE
12:     for edge $(v_i, v_j, B)$ in $oldOE(v_i)$ do
13:         for production $A ::= BC \in \mathcal{G}$ do
14:             for edge $(v_j, v_k, C)$ in $newOE(v_j)$ do
15:                 if edge $(v_i, v_k, A) \notin OE(v_i)$ then
16:                     add edge $(v_i, v_k, A)$ to $futureOE(v_i)$
17:                     $continue$ = TRUE
```

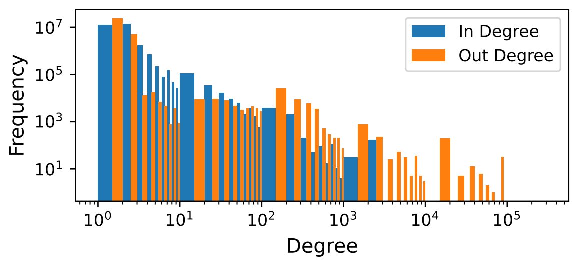


Figure 6: In-degree vs. Out-degree Distributions (the final data-flow graph from benchmark PostgreSQL)

This discrepancy can be attributed to the fact that although the total in-degrees and out-degrees of a graph are always identical, the distribution of in-degrees and out-degrees can differ significantly. In the example reported in Figure 6, the out-degree exhibits a more skewed distribution with more vertices with high (out)-degrees. In this case, the forward model may benefit from the improved spatial locality when accessing adjacency lists and thus tends to perform better.

However, it is difficult to leverage the above insight as the degree distribution of the final graph is unknown a priori. Luckily, we find that asymmetric degree distributions are linked to the left and right recursion of grammar rules: left recursion in the grammar results in increased out-degrees of vertices, while right recursion increases the in-degrees of vertices. Based on this rationale, the graph system can check the existence of left and right recursion in the grammar rules, and select the model direction accordingly.

Our second key observation is about the traversal order of "old" and "new" edge lists. Recall that with redundancy elimination, the system needs to examine three combinations of edge pairs: "old-new", "new-old", and "new-new".

***"old-new" vs. "new-old"**: Despite the symmetry between "old-new" and "new-old" traversal orders of edge pairs, their data locality can vary substantially.*

This discrepancy can be attributed to the fact the "old" edge list keeps accumulating as "new" edges turn into "old" ones over iterations, causing the "old" edge list significant larger than the "new" edge list for most of the time. In fact, we observe better performance if the processing traverses the (shorter) "new" edge list in the outer iteration and the (longer) "old" edge list in the inner iteration, likely due to the improved spatial locality in accessing the "old" edge list.

Now consider the model direction. For the forward and backward models, we find that the "new-old" traversal order can only be achieved partially. Taking the forward model as an example, as shown in Algorithm 6, the first major for-loop (Line 2) follows the "new-old" pattern, but the second major for-loop (Line 12) does not. Unfortunately, we cannot change their order without affecting the correctness.

In contrast, we find that the "new-old" optimization is fully applicable to the bidirectional model, as demonstrated in Algorithm 7. In this model, both the first and second major loop nests traverse the "new" edge list in the outer layer and the "old" edge list in the inner layer. This is feasible because, in the bidirectional model, the vertex being processed is positioned between a pair of adjacent edges, unlike in the forward or backward models, where the vertex is situated at one end of two adjacent edges (see Figure 2).

6.2 Parallelism

The complexity of parallel CFL reachability analysis arises from the dynamic nature of the graph. Parallel models must ensure the correctness of results while threads concurrently read from and update the graph. An intuitive parallelization of the vertex-centric models is to execute the vertex function on different vertices in parallel. However, our analysis reveals that this approach may not be safe for all model directions.

***Vertex-Level Parallelism**: The forward and backward models can safely run in parallel at the vertex level, whereas the bidirectional model introduces data races.*

In both forward and backward vertex-centric models, new edges are always added to the edge list of the vertex being processed v_i (see Figure 2a and 2b). This ensures that threads only modify the edge lists of their assigned vertices. However, in the bidirectional model, new edges are inserted into the edge lists of v_i's in/out-neighbors (see Figure 2c). This creates the possibility of multiple threads inserting edges into the same vertex's adjacency list, leading to data races. To avoid

Algorithm 7 f_{BI} with "new-old" Optimization

```
1: function $f_{BI}(v_i)$
2:     for edge $(v_j, v_i, B)$ in $newIE(v_i)$ do
3:         for production $A ::= B \in \mathcal{G}$ do
4:             if edge $(v_j, v_i, A) \notin OE(v_j)$ then
5:                 add edge $(v_j, v_i, A)$ to $futureOE(v_j)$
6:                 add edge $(v_j, v_i, A)$ to $futureIE(v_i)$
7:                 $continue$ = TRUE
8:         for production $A ::= BC \in \mathcal{G}$ do
9:             for edge $(v_i, v_k, C)$ in $oldOE(v_j) \cup newOE(v_j)$ do
10:                if edge $(v_j, v_k, A) \notin OE(v_j)$ then
11:                    add edge $(v_j, v_k, A)$ to $futureOE(v_j)$
12:                    add edge $(v_j, v_k, A)$ to $futureIE(v_k)$
13:                    $continue$ = TRUE
14:    for edge $(v_i, v_k, B)$ in $newOE(v_i)$ do
15:        for production $A ::= CB \in \mathcal{G}$ do
16:            for each edge $(v_j, v_i, C)$ in $oldIE(v_i)$ do
17:                if edge $(v_j, v_k, A) \notin OE(v_j)$ then
18:                    add edge $(v_j, v_k, A)$ to $futureOE(v_j)$
19:                    add edge $(v_j, v_k, A)$ to $futureIE(v_k)$
20:                    $continue$ = TRUE
```

Table 3: Trade-off between Locality and Parallelism

Models	Locality	Parallelism
V-centric (FW/BW)	🟥 "old-new" loop 🟩 out- or in-neighbor list	🟩 vertex-parallel
V-centric (BI)	🟩 "new-old" loops 🟥 out- and in-neighbor list	🟥 sync-required

data races, one may employ concurrent data structures, such as concurrent queues, which, however, comes with both time and memory overhead (see Section 8).

Similar data races occur in the worklist-based algorithm used in *POCR*, while *Graspan* avoids these races by adopting a model that is similar to our forward model.

6.3 Trade-offs

Based on the above discussions, we find no single direction outperforms the others in both parallism and data locality, as summarized in Table 3. The unidirectional models require only a single graph copy, whereas the bidirectional model needs both in- and out-neighbor adjacency lists, resulting in higher memory pressure. However, the bidirectional model supports the full "new-old" locality optimization, while the forward and backward models do not. In terms of parallelism, the unidirectional models are clearly superior. In Section 8, we will report the performance of these model variants.

7 Implementation

We implemented the above design ideas into a graph system for in-memory CFL reachability analysis, called *GraCFL*.

7.1 System Configuration

GraCFL is a customizable graph system that allows users to select the model configuration based on their specific needs and data characteristics. It supports the vertex-centric model in all three directions. For each model direction, it offers both serial and parallel execution (see Section 7.3). Furthermore, *GraCFL* allows users to choose between topology-driven and grammar-driven edge derivation, based on grammar's complexity. After discussing the performance in Section 8, we will offer some guidelines for model selection.

7.2 Data Structures

GraCFL employs a group of data structures customized to the proposed models and optimizations, as listed below.

Adjacency Lists. To represent the growing graph, *GraCFL* needs in-neighbor or/and out-neighbor adjacency lists.

For topology-driven models, it is often sufficient to use a 2D vector like `vector<vector<Edge>>`, where the outer level is indexed by the vertex IDs and the inner level is a vector of `Edge` that consists of a label and a neighbor's ID.

For grammar-driven models, *GraCFL* needs to access the edges of a vertex with a specific label (see Algorithm 6). To avoid scanning the neighbors, *GraCFL* uses a 3D vector:

```
vector<vector<vector<ull>>> edges
```

where the outer two levels are indexed by vertex ID and label ID, and the innermost level stores the IDs of neighbors. This data structure returns a list of neighbors associated with a given vertex ID and label in constant time.

Rule Lookup Tables. Topology-driven models need to check if a pair of labels matches any rule's RHS symbols, like $A ::= BC \in \mathcal{G}$ for edge pair BC, while grammar-driven models need to obtain all the rules (like $A ::= BC$) whose first (or second) RHS symbol matches the given symbol (e.g., B).

To avoid scanning the rules, *Graspan* uses a hashmap to find the LHS symbol based on the RHS symbol(s). However, the evaluation of the hash function can limit performance. Instead, we store the rules in tables, based on the assumption that grammars typically have a limited number of rules.

Consider rules $A ::= BC$ and $D ::= E$, *GraCFL* generates the following lookup tables:

- table `LHS1Table[`E`]` returns D;
- table `LHS2Table[getValue(`BC`)]` returns A;
- table `RHS1Table[`B`]` returns (A, C);
- table `RHS2Table[`C`]` returns (A, B).

When multiple rules match, these tables return a list. Each lookup takes (nearly) constant time.

Table 4: Graph Statistics of Benchmarks

Benchmarks		#Vertices	#Init. Edges	#New Edges	
				Self	Other
DF	httpd	5.7M	10.0M	-	9.3M
	postgres	29.8M	34.8M	-	21.3M
	linux	42.4M	44.0M	-	55.2M
PT	httpd	1.7M	3.0M	5.0M	896.2M
	postgres	5.2M	9.4M	15.6M	837.2M
	linux	11.3M	19.0M	33.8M	133.8M
JPT	mapreduce	21.9M	41.9M	-	57.3M
	hdfs	5.3M	10.2M	-	1.8B

Local Edge Hashset. For termination and efficiency, the graph system should prevent edge duplication—checking the existence of an edge before inserting it, like $(v_i, v_k, A) \notin E$.

First, storing the (sparse) graph in an adjacency matrix is impractical due to the sheer amount of vertices in real graphs. Instead, *Graspan* keeps edges sorted and removes duplicates in-between iterations [73]—a process that is computationally demanding. In contrast, *POCR* [30] utilizes a sparse bit vector to store edges of a vertex under a particular label, inherently preventing duplicates, but incurring additional overhead for edge membership checks and edge list traversal.

Unlike existing designs, *GraCFL* keeps a local hashset for each vertex and each label to check edge membership, which together form a 3D data structure:

```
vector<vector<unordered_set<ull>>> edges
```

where the first two vectors are for the source vertex and edge label, while the `unordered_set` stores destination vertices.

Together, the adjacency list and local edge hashset form a "dual graph" representation, which makes both edge list traversals and edge membership checks efficient, albeit at the cost of a larger memory footprint.

In addition, all major memory allocation in *GraCFL* is done via `jemalloc`, which is a general-purpose implementation of malloc(3) that emphasizes scalable concurrency support.

7.3 Parallelization

GraCFL uses the OpenMP library for parallel forward and backward models. For the bidirectional model, it employs a couple of concurrent data structures from the TBB library:

- `concurrent_vector` [17] for the adjacency lists
- `concurrent_unordered_set` [16] for the local hashset

As our evaluation will show, the use of these containers leads to increased memory consumption, thus their deployment should be approached with caution.

8 Evaluation

This section aims to answer two main research questions.

- **RQ1:** How well does *GraCFL* perform compared to state-of-the-art graph systems for CFL reachability?
- **RQ2:** How do previously discussed design choices and optimizations quantitatively impact performance?

8.1 Methodology

Experiment Setup. All of our experiments were conducted on a 16-core 2.10GHz Intel(R) Xeon(R) CPU E5-2683 v4 with 32 hyper-threads and 256GB memory. The machine runs Rocky Linux release 8.8 (Green Obsidian). All programs we evaluated, including the baselines, were compiled with the -O3 optimization flag, and their parallel versions were run using 32 threads unless otherwise noted.

Benchmarks. To evaluate the performance of *GraCFL*, we utilized graphs and grammars from the *Graspan* project [73], which include two points-to analysis grammars, one for C/C++ programs with 12 rules and one for Java programs [82] with 21 rules, as well as a single-rule data-flow analysis for C/C++ programs. The graphs are generated from codebases of Linux 4.4.0-rc5, PostgreSQL 8.3.9, Apache httpd 2.2.18, as well as Hadoop HDFS 2.0.3 and MapReduce 2.7.5. Table 4 lists the statistics of the initial and final graphs.

8.2 Overall Performance

We compared *GraCFL* with state-of-the-art (SOTA) systems for CFL reachability, including *Graspan* [73] and *POCR* [30]. For simplicity, we use FW, BW, BI to refer to the forward, backward, and bidirectional models in *GraCFL*, respectively.

Baseline Setups. Note that *Graspan* is designed for out-of-core processing scenarios. Here, we configured it so that the entire graph remains residing in memory during the whole processing. *Graspan* provides multiple implementations in different languages [4]. We chose *Graspan-C*, the fastest one among these implementations. As to *POCR* [5], to ensure it operates under its optimal performance conditions, we fed it with both *Graspan*- and *POCR*-style grammars when it is possible and chose the results with better performance. In addition, *Graspan* is configured to use all 32 CPU cores, while *POCR* ran on a single core for its single-threaded design.

Results Summary. Table 5 reports the execution times of two representative configurations of *GraCFL*: a FW model with 32 threads and a serial BI model, and the three SOTA systems following the above settings. In general, the parallel FW model of *GraCFL* performs the best across all tested benchmarks, achieving 2.44×-318.10× speedup over *Graspan-C* and 1.41×-57.60× speedup over *POCR*. In addition, the serial BI model of *GraCFL* outperforms SOTA systems for

[4]https://github.com/Graspan
[5]https://github.com/kisslune/POCR

Table 5: Comparison with State-of-the-art (SOTA) Systems

Benchmarks		**GraCFL**		**SOTA Systems**		**Speedups**			
				Graspan-C	**POCR**	**FW over**	**BI over**	**FW over**	**BI over**
		(FW, 32t)	**(BI, 1t)**	**(32t)**	**(1t)**	**Graspan-C**	**Graspan-C**	**POCR**	**POCR**
DF	httpd	**8s**	18s	34s	32s	4.25x	1.89x	4.00x	1.78x
	postgres	**93s**	268s	471s	131s	5.06x	1.76x	1.41x	0.49x
	linux	**127s**	425s	735s	209s	5.79x	1.73x	1.65x	0.49x
PT	httpd	**90s**	198s	4007s	2742s	44.52x	20.24x	30.47x	13.85x
	postgres	**104s**	254s	3607s	1740s	34.68x	14.20x	16.73x	6.85x
	linux	**43s**	190s	105s	319s	2.44x	0.55x	7.42x	1.68x
JPT	mapreduce	**17s**	96s	182s	194s	10.71x	1.90x	11.41x	2.02x
	hdfs	**82s**	248s	26084s	4723s	318.10x	105.18x	57.60x	19.04x

Table 6: Redundancy Elimination (GraCFL-BI)

Benchmarks		**Vector Copying**			**Sliding Pointers**		
		Com	**Upd**	**Total**	**Com**	**Upd**	**Total**
DF	httpd	24s	24s	48s	**11s**	**7s**	**18s**
	postgres	377s	467s	844s	**116s**	**152s**	**268s**
	linux	593s	721s	1314s	**190s**	**235s**	**425s**
PT	httpd	262s	177s	439s	**179s**	**19s**	**198s**
	postgres	485s	543s	1028s	**196s**	**58s**	**254s**
	linux	628s	962s	1590s	**101s**	**89s**	**190s**
JPT	mapreduce	345s	564s	909s	**50s**	**46s**	**96s**
	hdfs	319s	123s	442s	**239s**	**9s**	**248s**

most tested cases, faster than the 32-thread *Graspan-C* and serial *POCR* in seven and six out of eight cases, respectively.

8.3 Detailed Evaluation

Next, we assess different design aspects separately in detail.

Redundancy Elimination. Results in Table 6 indicate that the use of sliding pointers brings 1.78×-9.47× end-to-end speedup over the vector-copying approach under the BI model. Interestingly, not only is the edge group update time reduced, but also the computation time. Similar results were also observed on the FW and BW models. This is because the sliding pointers method uses a single vector to store all edges of a vertex, improving the locality, compared to the design using separate vectors for different edge groups.

Edge Derivation Strategy. Figure 7 compares the two edge derivation strategies under the BI model. The results show a dichotomy: the grammar-driven method excels in pointer analysis cases (over 20× speedup), except for "PT-linux", while the topology-driven one runs faster in all data-flow analysis cases and "PT-linux" (about 2× speedup). The above performance disparity may be linked to some input features.

- *Grammar complexity.* Simpler grammars imply fewer cases (RHS of rules) to check for a given edge pair, limiting the benefits of grammar-driven models.

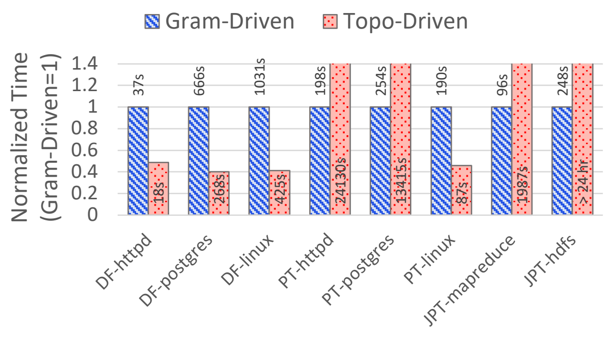


Figure 7: Edge Derivation Strategy (GraCFL-BI)

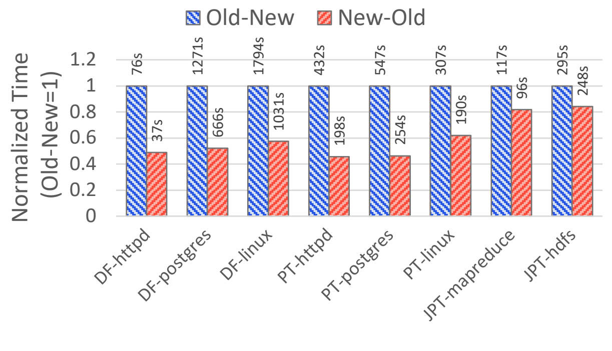


Figure 8: Data Locality (GraCFL-BI-gram-driven)

- *Degree distribution.* For vertices with relatively low degrees, grammar-driven method does not have much room for reducing "wasted checks"; on the other hand, it suffers from poor locality due to the use of more scattered 3D vector-based adjacency list.

For the above reasons, we chose to employ topology-driven method for data-flow analysis (with a single grammar rule) and grammar-driven approach for the two types of pointer analysis (with 12 and 21 rules).

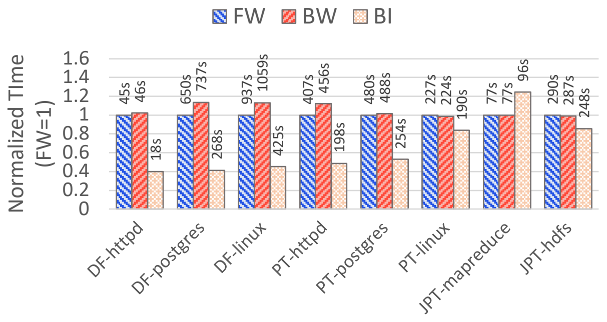


Figure 9: Serial Performance of GraCFL

Data Locality. First, Figure 8 compares the performance between serial BI models (gram-driven) with and without the "new-old" optimization—always traversing the new edge list in the outer loop and old edges in the inner loop (see Section 6.1). The results indicate consistent benefits with the "new-old" optimization, roughly cutting the time by half.

Figure 9 compares the serial performance among different model directions. In general, the BI model performs better than FW and BW models in seven out of eight cases, thanks to the application of "new-old" optimization. Unfortunately, as discussed earlier in Section 6.1, this optimization is not applicable to FW and BW models.

When comparing the FW and BW models, the FW model performs consistently better in the data-flow analysis. This is related to the recursive grammar rules. The two grammars for pointer analysis have both left and right recursions, while data-flow analysis only exhibits left recursion. To test this, we changed the only rule in data-flow analysis from left recursion to right recursion (i.e., $n ::= e\ n$). In this case, the BW model wins in two out of three bechmarks.

Parallelization. First, as discussed in Section 6.2, both FW and BW models are synchronization-free and can be run in parallel at the vertex-level. Figure 10 reports the speedup of the parallel FW model where the baseline is the case when the number of threads is set to one. The machine has a 16-core CPU with hyper-threading (i.e., 32 logical cores). While the trends indicate improved performance with more threads, the maximum speedup varies significantly across benchmarks, ranging from 4.71× to 7.38×.

In contrast, the parallel BI model relies on concurrent data structures (see Section 7.2) to handle potential simultaneous reads and writes to the same vertex's edge list. As shown in Figure 11, increasing the number of threads improves performance, but most benchmarks exhibit varying degrees of scalability degradation compared to the parallel FW model (Figure 10). In terms of the absolute execution time, the FW

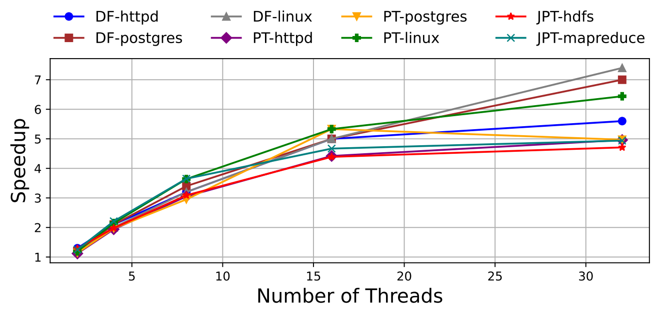


Figure 10: Parallel Performance of GraCFL-FW

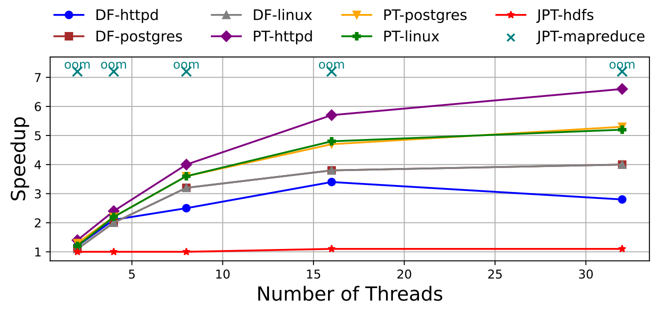


Figure 11: Parallel Performance of GraCFL-BI

Table 7: Peak Memory Usage (in GB)

Benchmarks		GraCFL			
		FW		BI	
		(1t)	(32t)	(1t)	(32t)
DF	httpd	**2.2**	2.3	2.8	11.6
	postgres	**9.7**	9.8	12.3	54.2
	linux	**14.7**	14.8	18.5	77.7
PT	httpd	**34.3**	34.5	40.8	69.0
	postgres	**37.6**	37.7	44.9	114.6
	linux	**22.7**	22.8	29.3	156.4
JPT	mapreduce	**58.1**	58.2	79.2	oom
	hdfs	**70.8**	**70.8**	86.5	215.1

model outperforms the BI model on all benchmarks except 'PT-httpd', with speedups ranging from 0.71× to 6.91×.

Furthermore, the use of concurrent data structures easily leads to a significant increase in memory usage, as reported in Table 7. Memory consumption increases by 2.00×-6.86× when using 32 threads for the BI model.

8.4 Guidelines for System Configuration

Based on our results, *GraCFL* can be configured considering computing resources and data characteristics.

Computing Resources. For environments with limited cores (e.g., fewer than 4), the serial BI model often delivers the best performance. Conversely, with higher core counts available, parallel FW and BW models are recommended for superior efficiency. In memory-constrained settings, it is advisable to avoid the BI model, especially its parallelized version, due to its higher memory consumption.

Data Characteristics. The input properties are also crucial for model configuration. In the presence of a large alphabet Σ, we recommend enabling grammar-driven edge derivation. When the grammar consists of only left or right recursion rules, select the FW or BW model accordingly. In addition, graph structures may also play a significant role, but their dynamic nature and complex interaction with grammar make practical leverage extremely challenging.

9 Related Work

Yannakakis introduced CFL reachability as a generalized form of graph reachability [78]. To solve this problem, Melski and Reps introduced a cubic algorithm [45]. After that, a subcubic algorithm was designed by Chaudhuri [6, 7, 48]. More recently, Koutris [24] offered an $\Omega(n^{2.5})$ lower bound for Dyck-2 reachability, a special case of CFL reachability.

Systems for CFL Reachability. Building on the "Big Data" perspective introduced by *Graspan* [73], several systems have been designed for CFL-reachability-based interprocedural program analysis. These include *BigSpa* [83], which utilizes Spark and Redis for cloud-based analysis; *Grapple* [84], a disk-based system for checking finite-state properties via alias and data-flow analysis; *Chianina* [86], which models flow-sensitive analysis as an evolving graph problem with support of disk-based processing; *BigDataflow* [70], which designs a distributed worklist algorithm targeting clusters; and *DStream* [74], an out-of-core streaming system for IFDS analysis, solved as CFL reachability problems [54].

Additionally, numerous systems have been developed for Android app analysis, such as *FlowDroid* [2], *DroidSafe* [14], *LiveDroid* [11], *DiskDroid* [34], and *SADroid* [33], to name a few. These systems are typically built on top of code analysis frameworks like *Soot* [72] and *WALA* [12], whose analyses are driven by CFL reachability-based tabulation solvers.

Alternatively, a Datalog engine, like *Soufflé* [59], *DLV* [32], and *SociaLite* [60] or recursive state machines (RSMs) [1, 6] may also be employed to solve analysis problems that are equivalent to CFL reachability problems. However, they require a slightly different form of problem formalization.

Algorithmic Optimizations. There is also growing effort devoted to algorithmic optimizations of CFL reachability. *Pearl* [63] batches the propagation of reachability relations to improve edge derivation efficiency. *STG* [62] decomposes the context-free grammar and adopts a staged processing strategy for better efficiency, while Skewed Tabulation [28] aims to avoid inserting unnecessary summary edges during solving IFDS problems. Some other works focus on graph simplication, including graph folding [31], which collapses nodes to reduce graph size, and cycle elimination [10, 29, 76], which detects collapsible cycles in the graph to accelerate the calculation of transitive closure.

Most of the above optimizations are orthogonal to the ideas proposed in this work and can potentially be integrated to further enhance *GraCFL*.

Dyck Reachability. In addition to general CFL reachability, various variants have been studied in the literature, with Dyck reachability being the most extensively explored [5, 23, 25, 36–39, 64, 71, 80, 81], which is often tailored to specific problem domains or requirements within program analysis.

Vertex-Centric Graph Processing. Originally developed for distributed platforms [43], vertex-centric graph processing has since been adapted to various platforms, which include in-memory systems [65], out-of-core [26], as well as GPU [22, 47] and out-of-GPU-memory [56] environments.

CFL Parsing. CFL reachability analysis can be viewed as CYK parsing [8, 21, 58, 79] over a graph. Recent system optimizations [18–20, 27, 35, 57] have been proposed for parsing large volumes of JSON—a widely used context-free language for data exchange and storage on the Web.

Insights from both the existing system design of graph processing frameworks and the algorithmic optimizations in CFL parsing could potentially inform new optimization strategies for graph systems addressing CFL reachability.

10 Conclusion

This work explores the design space of a vertex-centric graph system for CFL reachability, a classic problem in program analysis. Specifically, it defines a multi-directional model, where each direction provides a different perspective on the problem-solving process. By systematically examining the model in terms of computation redundancy, edge derivation strategy, locality, and parallelism, this work provides a series of new insights into optimizing the graph system. Based on these insights, it presents *GraCFL*, a high-performance graph system that significantly outperforms state-of-the-art graph systems for solving large-scale CFL reachability problems.

Acknowledgments

We thank all reviewers for their valuable comments and the authors of *Graspan* and *POCR* for making their source code and datasets publicly available. This work was supported in part by National Science Foundation grants CCF-2106383 and CCF-1813173 awarded to the UC, Riverside.

References

[1] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. 2001. Analysis of Recursive State Machines. In *Computer Aided Verification*, Gérard Berry, Hubert Comon, and Alain Finkel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 207–220.

[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

[3] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. 2017. To push or to pull: On reducing communication and synchronization in graph computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. 93–104.

[4] Sergey Brin and Lawrence Page. 1998. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems* 30, 1 (1998), 107–117. https://doi.org/10.1016/S0169-7552(98)00110-X Proceedings of the Seventh International World Wide Web Conference.

[5] Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. 2017. Optimal Dyck Reachability for Data-Dependence and Alias Analysis. *Proc. ACM Program. Lang.* 2, POPL, Article 30 (dec 2017), 30 pages. https://doi.org/10.1145/3158118

[6] Swarat Chaudhuri. 2008. Subcubic Algorithms for Recursive State Machines. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. Association for Computing Machinery, New York, NY, USA, 159–169. https://doi.org/10.1145/1328438.1328460

[7] Dmitry Chistikov, Rupak Majumdar, and Philipp Schepper. 2022. Subcubic certificates for CFL reachability. *Proc. ACM Program. Lang.* 6, POPL, Article 41 (jan 2022), 29 pages. https://doi.org/10.1145/3498702

[8] John Cocke and Jacob T. Schwartz. 1970. *Programming Languages and Their Compilers: Preliminary Notes*. Technical Report. Courant Institute of Mathematical Sciences, New York University. Unpublished manuscript.

[9] Shuo Ding and Qirun Zhang. 2023. Mutual Refinements of Context-Free Language Reachability. In *Static Analysis*, Manuel V. Hermenegildo and José F. Morales (Eds.). Springer Nature Switzerland, Cham, 231–258.

[10] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. 1998. Partial online cycle elimination in inclusion constraint graphs. *SIGPLAN Not.* 33, 5 (May 1998), 85–96. https://doi.org/10.1145/277652.277667

[11] Umar Farooq, Zhijia Zhao, Manu Sridharan, and Iulian Neamtiu. 2020. Livedroid: Identifying and preserving mobile app state in volatile runtime environments. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

[12] Stephen Fink and Eran Yahav. 2006. The WALA Framework for Static Analysis of Java Bytecode. In *Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE)*. ACM, 1–2. https://doi.org/10.1145/1134760.1134762

[13] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. {PowerGraph}: distributed {Graph-Parallel} computation on natural graphs. In *10th USENIX symposium on operating systems design and implementation (OSDI 12)*. 17–30.

[14] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. 2015. Information flow analysis of android applications in droidsafe.. In *NDSS*, Vol. 15. 110.

[15] Samuel Grossman, Heiner Litz, and Christos Kozyrakis. 2018. Making pull-based graph processing performant. *ACM SIGPLAN Notices* 53, 1 (2018), 246–260.

[16] Intel Corporation. 2024. concurrent_unordered_set Class. oneAPI Specification by Intel. https://spec.oneapi.io/versions/latest/elements/oneTBB/source/containers/concurrent_unordered_set_cls.html Accessed: 2024-06-07.

[17] Intel Corporation. 2024. concurrent_vector Class. oneAPI Specification by Intel. https://spec.oneapi.io/versions/latest/elements/oneTBB/source/containers/concurrent_vector_cls.html Accessed: 2024-06-07.

[18] Lin Jiang, Junqiao Qiu, and Zhijia Zhao. 2020. Scalable structural index construction for JSON analytics. *Proceedings of the VLDB Endowment* 14, 4 (2020), 694.

[19] Lin Jiang, Xiaofan Sun, Umar Farooq, and Zhijia Zhao. 2019. Scalable processing of contemporary semi-structured data on commodity parallel processors-a compilation-based approach. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 79–92.

[20] Lin Jiang and Zhijia Zhao. 2022. JSONSki: Streaming semi-structured data with bit-parallel fast-forwarding. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 200–211.

[21] Tadao Kasami. 1965. *An efficient recognition and syntax-analysis algorithm for context-free languages*. Technical Report AFCRL-65-758. Air Force Cambridge Research Laboratory, Bedford, MA.

[22] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. 2014. CuSha: vertex-centric graph processing on GPUs. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. 239–252.

[23] John Kodumal and Alex Aiken. 2004. The Set Constraint/CFL Reachability Connection in Practice. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation* (Washington DC, USA) *(PLDI '04)*. Association for Computing Machinery, New York, NY, USA, 207–218. https://doi.org/10.1145/996841.996867

[24] Paraschos Koutris and Shaleen Deep. 2023. The Fine-Grained Complexity of CFL Reachability. *Proc. ACM Program. Lang.* 7, POPL, Article 59 (jan 2023), 27 pages. https://doi.org/10.1145/3571252

[25] Shankaranarayanan Krishna, Aniket Lal, Andreas Pavlogiannis, and Omkar Tuppe. 2024. On-the-Fly Static Analysis via Dynamic Bidirected Dyck Reachability. *Proc. ACM Program. Lang.* 8, POPL, Article 42 (jan 2024), 30 pages. https://doi.org/10.1145/3632884

[26] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. {GraphChi}:{Large-Scale} graph computation on just a {PC}. In *10th USENIX symposium on operating systems design and implementation (OSDI 12)*. 31–46.

[27] Geoff Langdale and Daniel Lemire. 2019. Parsing gigabytes of JSON per second. *The VLDB Journal* 28, 6 (2019), 941–960.

[28] Yuxiang Lei, Camille Bossut, Yulei Sui, and Qirun Zhang. 2024. Context-Free Language Reachability via Skewed Tabulation. *Proc. ACM Program. Lang.* 8, PLDI, Article 221 (jun 2024), 24 pages. https://doi.org/10.1145/3656451

[29] Yuxiang Lei and Yulei Sui. 2019. Fast and Precise Handling of Positive Weight Cycles for Field-Sensitive Pointer Analysis. In *Static Analysis*, Bor-Yuh Evan Chang (Ed.). Springer International Publishing, Cham, 27–47.

[30] Yuxiang Lei, Yulei Sui, Shuo Ding, and Qirun Zhang. 2022. Taming transitive redundancy for context-free language reachability. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 1556–1582.

[31] Yuxiang Lei, Yulei Sui, Shin Hwei Tan, and Qirun Zhang. 2023. Recursive State Machine Guided Graph Folding for Context-Free Language Reachability. *Proc. ACM Program. Lang.* 7, PLDI, Article 119 (jun 2023),

25 pages. https://doi.org/10.1145/3591233

[32] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. 2006. The DLV System for Knowledge Representation and Reasoning. *ACM Trans. Comput. Logic* 7, 3 (jul 2006), 499–562. https://doi.org/10.1145/1149114.1149117

[33] Haofeng Li, Jie Lu, Haining Meng, Liqing Cao, Lian Li, and Lin Gao. 2024. Boosting the Performance of Multi-solver IFDS Algorithms with Flow-Sensitivity Optimizations. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 296–307.

[34] Haofeng Li, Haining Meng, Hengjie Zheng, Liqing Cao, Jie Lu, Lian Li, and Lin Gao. 2021. Scaling Up the IFDS Algorithm with Efficient Disk-Assisted Computing. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 236–247. https://doi.org/10.1109/CGO51591.2021.9370311

[35] Yinan Li, Nikos R Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. 2017. Mison: a fast JSON parser for data analytics. *Proceedings of the VLDB Endowment* 10, 10 (2017), 1118–1129.

[36] Yuanbo Li, Kris Satya, and Qirun Zhang. 2022. Efficient algorithms for dynamic bidirected Dyck-reachability. *Proc. ACM Program. Lang.* 6, POPL, Article 62 (jan 2022), 29 pages. https://doi.org/10.1145/3498724

[37] Yuanbo Li, Qirun Zhang, and Thomas Reps. 2021. On the complexity of bidirected interleaved Dyck-reachability. *Proc. ACM Program. Lang.* 5, POPL, Article 59 (jan 2021), 28 pages. https://doi.org/10.1145/3434340

[38] Yuanbo Li, Qirun Zhang, and Thomas Reps. 2022. Fast Graph Simplification for Interleaved-Dyck Reachability. *ACM Trans. Program. Lang. Syst.* 44, 2, Article 11 (may 2022), 28 pages. https://doi.org/10.1145/3492428

[39] Yuanbo Li, Qirun Zhang, and Thomas Reps. 2023. Single-Source-Single-Target Interleaved-Dyck Reachability via Integer Linear Programming. *Proc. ACM Program. Lang.* 7, POPL, Article 35 (jan 2023), 24 pages. https://doi.org/10.1145/3571228

[40] LLNL SAFIRE Contributors. 2025. CFL Andersen Alias Analysis Implementation in SAFIRE (LLVM 3.9.0). https://github.com/LLNL/SAFIRE/blob/master/llvm-3.9.0/lib/Analysis/CFLAndersAliasAnalysis.cpp. Accessed: 2025-01-14.

[41] LLVM Project Contributors. 2025. LLVM Alias Analysis Documentation. https://llvm.org/docs/AliasAnalysis.html. Accessed: 2025-01-14.

[42] LLVM Project Contributors. 2025. Unification-based Alias Analysis. https://github.com/llvm-mirror/llvm/blob/master/lib/Analysis/CFLSteensAliasAnalysis.cpp. Accessed: 2025-01-14.

[43] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.

[44] Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)* 48, 2 (2015), 1–39.

[45] David Melski and Thomas Reps. 2000. Interconvertibility of a class of set constraints and context-free-language reachability. *Theoretical Computer Science* 248, 1-2 (2000), 29–98.

[46] Ana Milanova. 2020. FlowCFL: generalized type-based reachability analysis: graph reduction and equivalence of CFL-based and type-based reachability. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 178 (nov 2020), 29 pages. https://doi.org/10.1145/3428246

[47] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: Transforming irregular graphs for gpu-friendly graph processing. *ACM SIGPLAN Notices* 53, 2 (2018), 622–636.

[48] Andreas Pavlogiannis. 2023. CFL/Dyck Reachability: An Algorithmic Perspective. *ACM SIGLOG News* 9, 4 (feb 2023), 5–25. https://doi.org/10.1145/3583660.3583664

[49] 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*. 126–135. https://doi.org/10.1109/CGO.2009.9

[50] Polyvios Pratikakis, Jeffrey S Foster, and Michael Hicks. 2006. Existential label flow inference via CFL reachability. In *Static Analysis: 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006. Proceedings 13*. Springer, 88–106.

[51] Jakob Rehof and Manuel Fähndrich. 2001. Type-base flow analysis: from polymorphic subtyping to CFL-reachability. *ACM SIGPLAN Notices* 36, 3 (2001), 54–66.

[52] Thomas Reps. 1995. Shape analysis as a generalized path problem. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program manipulation*. 1–11.

[53] Thomas Reps. 1998. Program analysis via graph reachability. *Information and Software Technology* 40, 11 (1998), 701–726. https://doi.org/10.1016/S0950-5849(98)00093-7

[54] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 49–61.

[55] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. 1994. Speeding up slicing. *ACM SIGSOFT Software Engineering Notes* 19, 5 (1994), 11–20.

[56] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. 2020. Subway: Minimizing data transfer during out-of-GPU-memory graph processing. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[57] Majid Saeedan, Ahmed Eldawy, and Zhijia Zhao. 2023. dsJSON: A Distributed SQL JSON Processor. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–25.

[58] Itiroo Sakai. 1962. Syntax in Universal Translation. In *Proceedings of the International Conference on Machine Translation of Languages and Applied Language Analysis*. Her Majesty's Stationery Office, London, 593–608.

[59] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On fast large-scale program analysis in datalog. In *Proceedings of the 25th International Conference on Compiler Construction*. 196–206.

[60] Jiwon Seo, Stephen Guo, and Monica S. Lam. 2013. SociaLite: Datalog extensions for efficient social network analysis. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 278–289. https://doi.org/10.1109/ICDE.2013.6544832

[61] Lei Shang, Xinwei Xie, and Jingling Xue. 2012. On-demand dynamic summary-based points-to analysis. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. 264–274.

[62] Chenghang Shi, Haofeng Li, Jie Lu, and Lian Li. 2024. Better Not Together: Staged Solving for Context-Free Language Reachability. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1112–1123.

[63] Chenghang Shi, Haofeng Li, Yulei Sui, Jie Lu, Lian Li, and Jingling Xue. 2023. Two Birds with One Stone: Multi-Derivation for Fast Context-Free Language Reachability Analysis. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 624–636. https://doi.org/10.1109/ASE56229.2023.00118

[64] Qingkai Shi, Yongchao Wang, Peisen Yao, and Charles Zhang. 2022. Indexing the Extended Dyck-CFL Reachability for Context-Sensitive Program Analysis. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 176 (oct 2022), 31 pages. https://doi.org/10.1145/3563339

[65] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM*

SIGPLAN symposium on Principles and practice of parallel programming. 135–146.

[66] Manu Sridharan and Rastislav Bodík. 2006. Refinement-based context-sensitive points-to analysis for Java. *ACM SIGPLAN Notices* 41, 6 (2006), 387–400.

[67] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven points-to analysis for Java. *ACM SIGPLAN Notices* 40, 10 (2005), 59–76.

[68] Yu Su, Ding Ye, and Jingling Xue. 2014. Parallel Pointer Analysis with CFL-Reachability. In *2014 43rd International Conference on Parallel Processing*. 451–460. https://doi.org/10.1109/ICPP.2014.54

[69] 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.

[70] Zewen Sun, Duanchen Xu, Yiyu Zhang, Yun Qi, Yueyang Wang, Zhiqiang Zuo, Zhaokang Wang, Yue Li, Xuandong Li, Qingda Lu, et al. 2023. BigDataflow: A Distributed Interprocedural Dataflow Analysis Framework. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1431–1443.

[71] Wensheng Tang, Dejun Dong, Shijie Li, Chengpeng Wang, Peisen Yao, Jinguo Zhou, and Charles Zhang. 2024. Octopus: Scaling Value-Flow Analysis via Parallel Collection of Realizable Path Conditions. *ACM Trans. Softw. Eng. Methodol.* 33, 3, Article 66 (mar 2024), 33 pages. https://doi.org/10.1145/3632743

[72] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. 214–224.

[73] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. 2017. Graspan: A Single-Machine Disk-Based Graph System for Interprocedural Static Analyses of Large-Scale Systems Code. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 389–404. https://doi.org/10.1145/3037697.3037744

[74] Xizao Wang, Zhiqiang Zuo, Lei Bu, and Jianhua Zhao. 2023. DStream: A Streaming-Based Highly Parallel IFDS Framework. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2488–2500. https://doi.org/10.1109/ICSE48619.2023.00208

[75] Guoqing Xu, Atanas Rountev, and Manu Sridharan. 2009. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *ECOOP 2009–Object-Oriented Programming: 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings 23*. Springer, 98–122.

[76] Pei Xu, Yuxiang Lei, Yulei Sui, and Jingling Xue. 2024. Iterative-Epoch Online Cycle Elimination for Context-Free Language Reachability. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 145 (apr 2024), 26 pages. https://doi.org/10.1145/3649862

[77] Carl Yang, Aydın Buluç, and John D Owens. 2018. Implementing push-pull efficiently in GraphBLAS. In *Proceedings of the 47th International Conference on Parallel Processing*. 1–11.

[78] Mihalis Yannakakis. 1990. Graph-Theoretic Methods in Database Theory. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Nashville, Tennessee, USA) *(PODS '90)*. Association for Computing Machinery, New York, NY, USA, 230–242. https://doi.org/10.1145/298514.298576

[79] Daniel H. Younger. 1967. Recognition and parsing of context-free languages in time n∧3. *Information and Control* 10, 2 (1967), 189–208.

[80] Hao Yuan and Patrick Eugster. 2009. An Efficient Algorithm for Solving the Dyck-CFL Reachability Problem on Trees. In *Programming Languages and Systems*, Giuseppe Castagna (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 175–189.

[81] Qirun Zhang, Michael R Lyu, Hao Yuan, and Zhendong Su. 2013. Fast algorithms for Dyck-CFL-reachability with applications to alias analysis. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 435–446.

[82] Xin Zheng and Radu Rugina. 2008. Demand-driven alias analysis for C. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 197–208.

[83] Zhiqiang Zuo, Rong Gu, Xi Jiang, Zhaokang Wang, Yihua Huang, Linzhang Wang, and Xuandong Li. 2019. BigSpa: An efficient interprocedural static analysis engine in the cloud. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 771–780.

[84] Zhiqiang Zuo, John Thorpe, Yifei Wang, Qiuhong Pan, Shenming Lu, Kai Wang, Guoqing Harry Xu, Linzhang Wang, and Xuandong Li. 2019. Grapple: A graph system for static finite-state property checking of large-scale systems code. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–17.

[85] Zhiqiang Zuo, Kai Wang, Aftab Hussain, Ardalan Amiri Sani, Yiyu Zhang, Shenming Lu, Wensheng Dou, Linzhang Wang, Xuandong Li, Chenxi Wang, and Guoqing Harry Xu. 2021. Systemizing Interprocedural Static Analysis of Large-scale Systems Code with Graspan. *ACM Trans. Comput. Syst.* 38, 1–2, Article 4 (jul 2021), 39 pages. https://doi.org/10.1145/3466820

[86] Zhiqiang Zuo, Yiyu Zhang, Qiuhong Pan, Shenming Lu, Yue Li, Linzhang Wang, Xuandong Li, and Guoqing Harry Xu. 2021. Chianina: An evolving graph system for flow-and context-sensitive analyses of million lines of C code. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 914–929.