Pipelining Bottom-up Data Flow Analysis

Qingkai Shi
The Hong Kong University of Science and Technology
Hong Kong, China
qshiaa@cse.ust.hk

Charles Zhang
The Hong Kong University of Science and Technology
Hong Kong, China
charlesz@cse.ust.hk

ABSTRACT

Bottom-up program analysis has been traditionally easy to parallelize because functions without caller-callee relations can be analyzed independently. However, such function-level parallelism is significantly limited by the calling dependence - functions with caller-callee relations have to be analyzed sequentially because the analysis of a function depends on the analysis results, a.k.a., function summaries, of its callees. We observe that the calling dependence can be relaxed in many cases and, as a result, the parallelism can be improved. In this paper, we present Coyote, a framework of bottom-up data flow analysis, in which the analysis task of each function is elaborately partitioned into multiple sub-tasks to generate pipelineable function summaries. These sub-tasks are pipelined and run in parallel, even though the calling dependence exists. We formalize our idea under the IFDS/IDE framework and have implemented an application to checking null-dereference bugs and taint issues in C/C++ programs. We evaluate Coyote on a series of standard benchmark programs and open-source software systems, which demonstrates significant speedup over a conventional parallel design.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**.

KEYWORDS

Compositional program analysis, modular program analysis, bottom-up analysis, data flow analysis, IFDS/IDE.

ACM Reference Format:
Qingkai Shi and Charles Zhang. 2020. Pipelining Bottom-up Data Flow Analysis. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3377811.3380425

1 INTRODUCTION

Bottom-up analyses work by processing the call graph of a program upwards from the leaves – before analyzing a function, all its callee functions are analyzed and summarized as function summaries [4, 8, 9, 11, 15, 16, 54, 63, 64]. These analyses have two key strengths: the

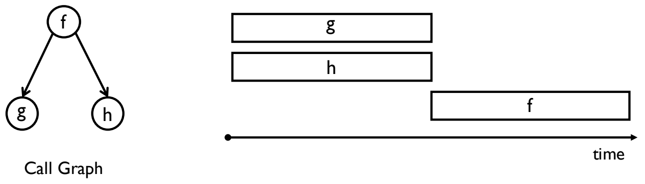


Figure 1: Conventional parallel design of bottom-up program analysis. Each rectangle represents the analysis task for a function.

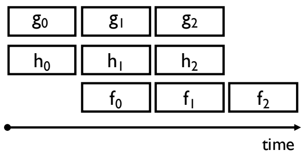


Figure 2: The analysis task of each function is partitioned into multiple sub-tasks. All sub-tasks are pipelined.

function summaries they compute are highly reusable and they are easy to parallelize because the analyses of functions are decoupled.

While almost all existing bottom-up analyses take advantage of such function-level parallelization, there is little progress in improving its parallelism. As reported by recent studies, it still needs to take a few hours, even tens of hours, to precisely analyze large-scale software. For example, it takes 6 to 11 hours for Saturn [64] and Calysto [4] to analyze programs of 685KLoC [4]. It takes about 5 hours for Pinpoint [54] to analyze about 8 million lines of code. With regard to the performance issues, McPeak et al. [35] pointed out that the parallelism often drops off at runtime and, thus, the CPU resources are usually not well utilized. Specifically, this is because the parallelism is significantly limited by the *calling dependence* – functions with caller-callee relations have to be analyzed sequentially because the analysis of a caller function depends on the analysis results, i.e., function summaries, of its callee functions. To illustrate this phenomenon, let us consider the call graph in Figure 1 where the function f calls the functions, g and h. In a conventional bottom-up analysis, only functions without caller-callee relations, e.g., the function g and the function h, can be analyzed in parallel. The analysis of the function f cannot start until the analyses of the functions, g and h, complete. Otherwise, when analyzing a call site of the function, g or h, in the function f, we may miss some effects of the callees due to the incomplete analysis.[1]

[1]This is different from a top-down method that can let the analysis of the function f run first but stop to wait for the analysis results of the function g when analyzing a call statement calling the function g.

In this paper, we present Coyote, a framework of bottom-up data flow analysis that breaks the limits of function boundaries, so that functions having calling dependence can be analyzed in parallel. As a result, we can achieve much higher parallelism than the conventional parallel design of bottom-up analysis. Our key insight is that many analysis tasks of a caller function only depend on partial analysis results of its callee functions. Thus, the analysis of the caller function can start before the analyses of its callee functions complete. Therefore, our basic idea is to partition the analysis task of a function into multiple sub-tasks, so that we can pipeline the sub-tasks to generate function summaries. The key to the partition is a soundness criterion, which requires a sub-task only depends on the summaries produced by the sub-tasks finished in the callees. Violating this criterion will cause the analysis to neglect certain function effects and make the analysis unsound.

To illustrate, assume that the analysis task of each function in Figure 1, e.g., the function f, is partitioned into three sub-tasks, f_0, f_1, and f_2, each of which generates one kind of function summaries. These sub-tasks satisfy the constraints that the sub-task f_i only depends on the function summaries produced by the sub-task g_j and the sub-task h_j ($j \leq i$). As a result, these sub-tasks can be pipelined as illustrated in Figure 2, where the analysis of the function f starts immediately after the sub-tasks g_0 and h_0 finish. Clearly, the parallelism in Figure 2 is much higher than that in Figure 1, providing a significant speedup over the conventional parallel design of bottom-up analysis.

In this paper, we formalize our idea under the IFDS/IDE framework for a wide range of data flow problems known as the inter-procedural finite distributive subset or inter-procedural distributive environment problems [45, 50]. In both problems, the data flow functions are required to be distributive over the merge operator. Although this is a limitation in some cases, the IFDS/IDE framework has been widely used for many practical problems such as secure information flow [3, 23, 42], typestate [19, 39], alias sets [40], specification inference [55], and shape analysis [47, 65]. Given any of those IFDS/IDE problems, conventional solutions compute function summaries either in a bottom-up fashion (e.g., [49, 67]) or in a top-down manner (e.g., [45, 50]), depending on their specific design goals. In this paper, we focus on the bottom-up solutions and aim to improve their performance via the pipeline parallelization strategy.

We implemented Coyote to path-sensitively check null dereferences and taint issues in C/C++ programs. Our evaluation of Coyote is based on standard benchmark programs and many large-scale software systems, which demonstrates that the calling dependence significantly limits the parallelism of bottom-up data flow analysis. By relaxing this dependence, our pipeline strategy achieves 2×-3× speedup over the conventional parallel design of bottom-up analysis. Such speedup is significant enough to make many overly lengthy analyses useful in practice. In summary, the main contributions of this paper include the following:

- We propose the design of pipelineable function summaries, which enables the pipeline parallelization strategy for bottom-up data flow analysis.
- We formally prove the correctness of our approach and apply it to a null analysis and a taint analysis to show its generalizability.

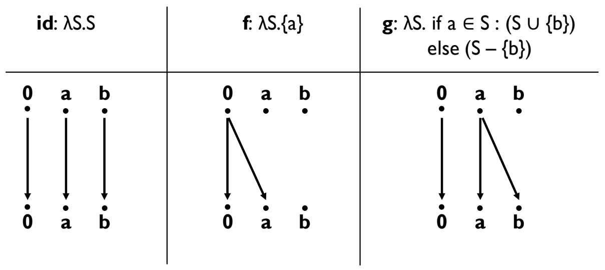


Figure 3: Data flow functions and their representation in the exploded super-graph [45].

- We conduct a systematic evaluation to demonstrate that our approach can achieve much higher parallelism and, thus, runs faster than the state of the arts.

2 BACKGROUND AND OVERVIEW

In this section, we introduce the background of the IFDS/IDE framework (Section 2.1) and provide an example to illustrate how we improve the parallelism of a bottom-up analysis by partitioning the analysis of a function (Section 2.2).

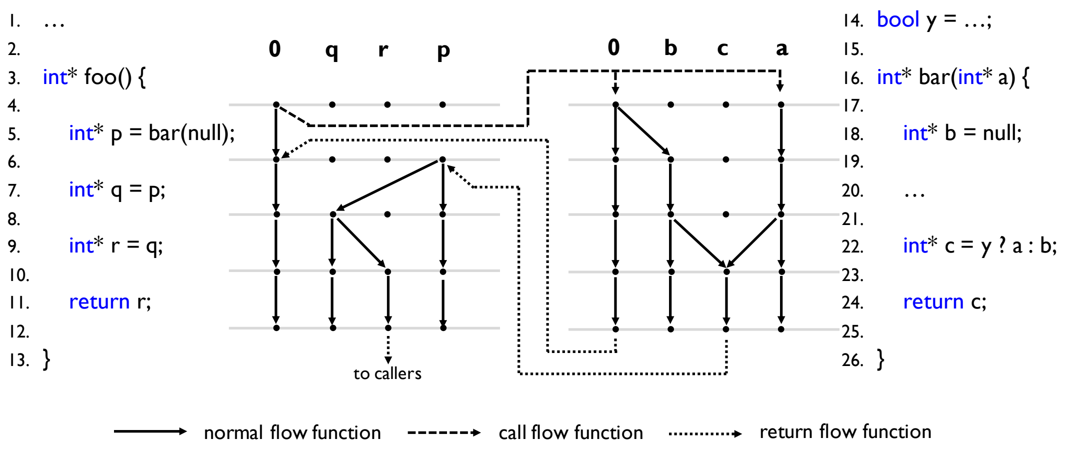
2.1 The IFDS/IDE Framework

The IFDS/IDE framework aims to solve a wide range of data flow problems known as Inter-procedural Finite Distributive Subset or Inter-procedural Distributive Environment problems [45, 50]. Its basic idea is to transform a data flow problem to a graph reachability problem on the *exploded super-graph*, which is built based on the inter-procedural control flow graph of a program.

The IFDS Framework. In the IFDS framework, every vertex $(s_i, \mathbf{d})$ in the exploded super-graph stands for a statically decidable data flow fact, or simply, fact, $\mathbf{d}$ at a program point s_i. Every edge models the data flow functions between data flow facts. In the paper, to ease the explanation, we use s_i to denote the program point at Line i in the code. For example, in an analysis to check null dereference, the vertex $(s_i, \mathbf{d})$ could denote that the variable d is a null pointer at Line i. As for the edges or data flow functions, Figure 3 illustrates three examples that show how the commonly-used data flow functions are represented as edges in the exploded super-graph. The vertices at the top are the data flow facts before a program point and the vertices at the bottom represent the facts after the program point.

The first data flow function **id** is the identity function which maps each data flow fact before a program point to itself. It indicates that the statement at the program point has no impacts on the data flow analysis.

The special vertex for the fact **0** is associated with every program point in the program. It denotes a tautology, a data flow fact that always holds. An edge from the fact **0** to a non-**0** fact indicates that the non-**0** fact is freshly created. For example, in the second function in Figure 3, the fact **a** is created, which is represented by an edge from the fact **0** to the fact **a**. At the same time, since **a** is the only fact after the data flow function, there is no edge connecting the fact **b** before and after the program point.



```
1.  ...
2.
3.  int* foo() {
4.
5.      int* p = bar(null);
6.
7.      int* q = p;
8.
9.      int* r = q;
10.
11.     return r;
12.
13. }
```

```
14. bool y = ...;
15.
16. int* bar(int* a) {
17.
18.     int* b = null;
19.
20.     ...
21.
22.     int* c = y ? a : b;
23.
24.     return c;
25.
26. }
```

Figure 4: An example of the exploded super-graph for a null-dereference analysis.

The third data flow function is a typical function that models the assignment $b = a$. In the exploded super-graph, the variable a has the same value as before. Thus, there is an edge from the data flow fact **a** to itself. The variable b gets the value from the variable a, which is modeled by the edge from the fact **a** to the fact **b**.

It is noteworthy that the data flow facts are not limited to simple values like the local variables in the examples of the paper. For example, in alias analysis, the facts can be sets of access paths [59]. In typestate analysis, the facts can be the combination of different typestates [39].

Figure 4 illustrates the exploded super-graph for a data flow analysis that tracks the propagation of null pointers. Since Line 18 assigns a null pointer to the variable b, we have the edge from the vertex $(s_{17}, \mathbf{0})$ to the vertex $(s_{19}, \mathbf{b})$, meaning that we have the data flow fact b = null at Line 19. Since Line 19 does not change the value of the variable a, we have the edge from the vertex $(s_{17}, \mathbf{a})$ to the vertex $(s_{19}, \mathbf{a})$, which means the data flow fact about the variable a does not change.

Assuming that s_{main} is the program entry point, the IFDS framework aims to find paths, or determine the reachability relations, between the vertex $(s_{\text{main}}, \mathbf{0})$ and the vertices of interests. Each of such paths represents that some data flow fact holds at a program point. For instance, the path from the vertex $(s_4, \mathbf{0})$ to the vertex $(s_{12}, \mathbf{r})$ in Figure 4 implies that the fact r = null holds at Line 12.

The IFDS method is efficient because it computes function summaries only once for each function. Each summary is a path on the exploded super-graph connecting a pair of vertices at the entry and the exit of a function. The path from the vertex $(s_{17}, \mathbf{a})$ to the vertex $(s_{25}, \mathbf{c})$ in Figure 4 is such a summary of the function bar. When analyzing the callers of the function bar, e.g., the function foo, we can directly jump from the vertex $(s_4, \mathbf{0})$ to the vertex $(s_6, \mathbf{p})$ using the summary without analyzing the function bar again.

The IDE Framework. The IDE framework is a generalization of the IFDS framework [50]. Similar to the IFDS framework, it also works as a graph traversal on an exploded super-graph. There are three major differences. First, each vertex on the exploded super-graph is no longer associated with a simple data flow fact **d**, but an environment mapping a fact **d** to a value **v** from a separate value domain, denoted as $\{\mathbf{d} \mapsto \mathbf{v}\}$. Second, due to the first difference, the data flow functions, i.e., the edges on the exploded super-graph, transform an environment $\{\mathbf{d} \mapsto \mathbf{v}\}$ to the other $\{\mathbf{d'} \mapsto \mathbf{v'}\}$. The third important difference is that each edge on the exploded super-graph is labeled with an environment transform function, which makes IDE no longer only a simple graph reachability problem. Instead, it has to find the paths between two vertices of interests and, meanwhile, compose the environment transform functions labeled on the edges along the paths. These differences widen the class of problems that can be expressed in the IFDS framework.

In this paper, for simplicity, we describe our work under the IFDS framework. This does not lose the generality for the IDE problems because, intuitively, both problems are solved by a graph traversal on the exploded super-graph.

2.2 Coyote in a Nutshell

Let us briefly explain our approach using the code and its corresponding exploded super-graph in Figure 4, where the analysis aims to track the propagation of null pointers.

Bottom-up Analysis. For the example in Figure 4, a conventional bottom-up analysis firstly analyzes the function bar and produces function summaries to summarize its behavior. With the function summaries in hand, the function foo then is analyzed.

Using the symbol $\rightsquigarrow$ to denote a path between two vertices, a common IFDS/IDE solution will generate the following two intraprocedural paths as the summaries of the function bar:

- The path $(s_{17}, \mathbf{a}) \rightsquigarrow (s_{25}, \mathbf{c})$ summarizes the function behavior that a null pointer created in a caller of the function bar, i.e., a = null, may be returned back to the caller.
- The path $(s_{17}, \mathbf{0}) \rightsquigarrow (s_{25}, \mathbf{c})$ summarizes the function behavior that a null pointer created in the function bar may be returned to the caller functions.

Note that we do not need to summarize the path $(s_{17}, \mathbf{0}) \rightsquigarrow (s_{25}, \mathbf{0})$ for the function bar, because the fact $\mathbf{0}$ is a tautology and always holds.

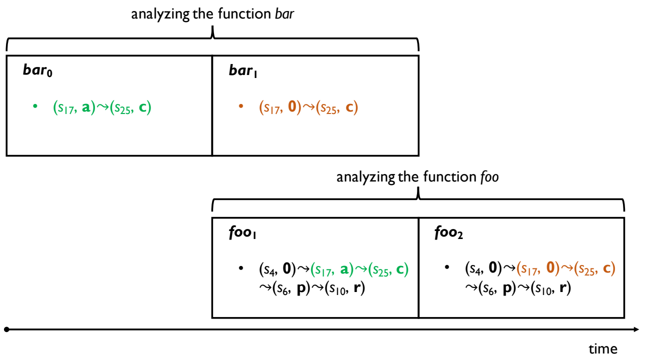

analyzing the function *bar*
*bar*₀
• (s_{17}, **a**)⇝(s_{25}, **c**)
*bar*₁
• (s_{17}, **0**)⇝(s_{25}, **c**)
analyzing the function *foo*
*foo*₁
• (s_4, **0**)⇝(s_{17}, **a**)⇝(s_{25}, **c**) ⇝(s_6, **p**)⇝(s_{10}, **r**)
*foo*₂
• (s_4, **0**)⇝(s_{17}, **0**)⇝(s_{25}, **c**) ⇝(s_6, **p**)⇝(s_{10}, **r**)
time

Figure 5: The pipeline parallelization strategy.

Next, we analyze the function foo by a graph traversal from the vertex $(s_4, \mathbf{0})$, which aims to track the propagation of null pointers and produce function summaries of the function foo. During the graph traversal, when the call flow functions (i.e., the dashed edges) are visited, we apply the summaries of the function bar and produce two summaries of the function foo as following ($[\![\cdot]\!]_{\mathsf{bar}}$ are the summaries of the function bar):

- The path $(s_4, \mathbf{0}) \rightsquigarrow [\![(s_{17}, \mathbf{a}) \rightsquigarrow (s_{25}, \mathbf{c})]\!]_{\mathsf{bar}} \rightsquigarrow (s_6, \mathbf{p}) \rightsquigarrow (s_{12}, \mathbf{r})$ summarizes the function behavior that a null pointer in the function foo will be returned to its callers.
- The path $(s_4, \mathbf{0}) \rightsquigarrow [\![(s_{17}, \mathbf{0}) \rightsquigarrow (s_{25}, \mathbf{c})]\!]_{\mathsf{bar}} \rightsquigarrow (s_6, \mathbf{p}) \rightsquigarrow (s_{12}, \mathbf{r})$ summarizes the function behavior that a null pointer in the callees of the function foo will be returned to the callers of the function foo.

Our Approach. As discussed before, in a conventional bottom-up analysis, the analysis of a caller function needs to wait for the analysis of its callees to complete. Differently, Coyote aims to improve the parallelism by starting the analysis of the function foo before completing the analysis of the function bar. To this end, Coyote partitions the analysis of each function f into three parts based on where a data flow fact is created. Such a partition categorizes the function summaries into three groups, f_0, f_1, and f_2, which we refer to as the pipelineable summaries:

- f_0 summarizes the behavior that some data flow facts created in the caller functions will be propagated back to the callers through the current function. The first summary of the function bar is an example.
- f_1 summarizes the behavior that some data flow facts created in the current function will be propagated back to the caller functions. The second summary of the function bar and the first summary of the function foo are two examples.
- f_2 summarizes the behavior that some data flow facts created in the callees are propagated to the current function and will continue to be propagated to the caller functions. The second summary of the function foo is an example.

According to the partition method, the summaries of the function foo is partitioned into two sets, foo_1 and foo_2, just as illustrated in Figure 5. Since the function foo does not have any function parameters, the set foo_0 is empty and, thus, omitted. Similarly, the summaries of the function bar is partitioned into two sets, bar_0 and bar_1. Since the function bar does not have any callees, the set bar_2 is empty and, thus, omitted. As detailed later, the above partition is sound because it satisfies the constraint that summaries in the set foo_i only depends on the summaries in the set $\mathsf{bar}_j (j \le i)$. Thus, we can safely pipeline the analyses of the function foo and the function bar - we can start analyzing the function foo immediately after summaries in the set bar_0 are generated.

In the remainder of this paper, under the IFDS framework, we formally present how to partition the analysis of a function to generate pipelineable function summaries, so that the parallelism of bottom-up analysis can be improved in a sound manner.

3 COYOTE: PIPELINED BOTTOM-UP ANALYSIS

To explain our method in detail, we first define the basic notations and terminologies in Section 3.1 and then explain the criteria that guide our partition method in Section 3.2. Based on the criteria, we present the technical details of our pipeline parallelization strategy from Section 3.3 to Section 3.5.

3.1 Preliminaries

To clearly present our approach, we introduce the following notations and terminologies.

Program Model. Given an IFDS problem, a program is modeled as an exploded super graph G that consists of a set of intra-procedural graphs $\{G_{\mathsf{f}}, G_{\mathsf{g}}, G_{\mathsf{h}}, \dots\}$ of the functions $\{\mathsf{f}, \mathsf{g}, \mathsf{h}, \dots\}$. Given a function f, its local graph G_{f} is a tuple $(L_{\mathsf{f}}, e_{\mathsf{f}}, x_{\mathsf{f}}, D_{\mathsf{f}}, E_{\mathsf{f}})$:

- L_{f} is the set of program locations in the function.
- $e_{\mathsf{f}}, x_{\mathsf{f}} \in L_{\mathsf{f}}$ are the entry and exit points of the function.
- D_{f} is the set of data flow facts in the function.
- $L_{\mathsf{f}} \times D_{\mathsf{f}}$ is the set of vertices of the graph.
- $E_{\mathsf{f}} \subseteq (L_{\mathsf{f}} \times D_{\mathsf{f}}) \times (L_{\mathsf{f}} \times D_{\mathsf{f}})$ is the edge set (see Figure 3).

As illustrated in Figure 4, the local graphs of different functions are connected by call and return flow functions, respectively.

Function Summaries. For any function f, its function summaries are a set of paths between data flow facts at the entry point and data flow facts at its exit point [45], denoted as $S_{\mathsf{f}} = \{(e_{\mathsf{f}}, \mathbf{a}) \rightsquigarrow (x_{\mathsf{f}}, \mathbf{b}) : \mathbf{a}, \mathbf{b} \in D_{\mathsf{f}}\}$. Apparently, we can generate these summaries by traversing the graph G_{f} from every vertex at the function entry.

Owing to function calls in a program, the summaries of a function often depend on the summaries of its callees. We say a summary set S depends on the other summary set S' if and only if there exists a path in the set S that subsumes a path in the set S'. As illustrated in Section 2.2, the summaries of the function foo depend on the summaries of the function bar.

Summary Dependence Graph. To describe the dependence between summary sets, we define the summary dependence graph, where a vertex is a set of function summaries and a directed edge indicates the source summary set depends on the destination summary set.

The summary dependence graph is built based on the call graph. Conventionally, vertices of the summary dependence graph are the summary sets $\{S_{\mathsf{f}}, S_{\mathsf{g}}, S_{\mathsf{h}} \dots\}$, and an edge from the summary set S_f to the summary set S_g exists if and only if the function f calls the function g. A bottom-up analysis works by processing the summary dependence graph upwards from the leaves. It starts generating summaries in a summary set if it does not depend on other summary sets or the summary sets it depends on have been generated. Summary sets that do not have dependence relations can be generated in parallel.

Problem Definition. In this paper, we aim to find a partition for the summary set of each function, say $\Pi(S_\mathrm{f}) = \{S_\mathrm{f}^0, S_\mathrm{f}^1, S_\mathrm{f}^2, \dots\}$,[2] such that a vertex of the summary dependence graph is no longer a complete summary set S_f but a subset S_f^i ($i \geq 0$). Meanwhile, to improve the parallelism, the bottom-up analysis based on the dependence graph should be able to generate summaries for a pair of caller and callee functions at the same time. In detail, the partition needs to satisfy the criteria discussed in the next subsection.

3.2 Partition Criteria

Given a pair of functions where the function f calls the function g, we use the set $\Omega(S_\mathrm{f}, S_\mathrm{g}) \subseteq \Pi(S_\mathrm{f}) \times \Pi(S_\mathrm{g})$ to denote the dependence relations between summary sets. Generally, an effective partition method must meet the following criteria to improve the parallelism of a bottom-up analysis.

The Effectiveness Criterion. This criterion concerns whether the dependence between summary sets in the conventional bottom-up analysis is actually relaxed, so that the parallelism can be improved. We say the partition is effective if and only if $|\Omega(S_\mathrm{f}, S_\mathrm{g})| < |\Pi(S_\mathrm{f}) \times \Pi(S_\mathrm{g})|$. Intuitively, this means that some summaries in the caller function do not depend on all summaries in callee functions. Thus, the dependence relation in the conventional bottom-up analysis is relaxed.

The Soundness Criterion. This criterion concerns the correctness after the dependence between summary sets is relaxed. We say the partition is sound if and only if the following condition is satisfied: if the set S_f^i depends on the set S_g^j, then $(S_\mathrm{f}^i, S_\mathrm{g}^j) \in \Omega(S_\mathrm{f}, S_\mathrm{g})$. Violating this criterion will cause the analysis to neglect certain function summaries and make the analysis unsound.

The Efficiency Criterion. This criterion concerns how many computational resources we need to consume in order to determine how to partition a summary set. Since summaries in the summary sets, S_f and S_g, are unknown before an analysis completes, the exact dependence relations between summaries in the two sets are also undiscovered. This fact makes it difficult to perform a fine-grained partition, unless the analysis has been completed and we have known what summaries are generated for each function.

As a trade-off, conventional bottom-up analysis does not partition the summary sets (or equivalently, $\Pi(S_\mathrm{f}) = \{S_\mathrm{f}\}$ and $\Pi(S_\mathrm{g}) = \{S_\mathrm{g}\}$). It conservatively utilizes the observation that all summaries in the set S_f may depend on certain summaries in the set S_g, i.e., $\Omega(S_\mathrm{f}, S_\mathrm{g}) = \{(S_\mathrm{f}, S_\mathrm{g})\}$. Such a conservative method satisfies the soundness criterion and does not partition the summary sets. However, apparently, it does not meet the effectiveness criterion because $|\Omega(S_\mathrm{f}, S_\mathrm{g})| = |\Pi(S_\mathrm{f}) \times \Pi(S_\mathrm{g})| = 1$.

[2]A set partition needs to satisfy $\cup_{i \geq 0} S_\mathrm{f}^i = S_\mathrm{f}$ and $\forall i, j \geq 0 : S_\mathrm{f}^i \cap S_\mathrm{f}^j = \emptyset$.

3.3 Pipelineable Summary-Set Partition

Generally, it is challenging to partition a summary set satisfying the above criteria because the exact dependence between summaries are unknown before the summaries are generated. We now present a coarse-grained partition method that requires few precomputations, and thus, meets the efficiency criterion. Meanwhile, it also meets the effectiveness and soundness criteria and, thus, can soundly improve the parallelism of a bottom-up analysis. We also establish a few lemmas to prove the correctness of our approach.

Intuitively, given a summary set S_f, we partition it according to where a data flow fact is created: in a caller of the function f, in the current function f, and in a callee of the function f. Formally, $\Pi(S_\mathrm{f}) = \{S_\mathrm{f}^0, S_\mathrm{f}^1, S_\mathrm{f}^2\}$, where

$$S_\mathrm{f}^0 = \{(e_\mathrm{f}, \mathbf{a}) \rightsquigarrow (x_\mathrm{f}, \mathbf{b}) : \mathbf{a} \neq \mathbf{0}\}$$
$$S_\mathrm{f}^1 = \{(e_\mathrm{f}, \mathbf{0}) \rightsquigarrow (e_\mathrm{g}, \mathbf{a}) \rightsquigarrow (x_\mathrm{f}, \mathbf{b}) : \mathrm{f} = \mathrm{g} \vee \mathbf{a} \neq \mathbf{0}\}$$
$$S_\mathrm{f}^2 = \{(e_\mathrm{f}, \mathbf{0}) \rightsquigarrow (e_\mathrm{g}, \mathbf{0}) \rightsquigarrow (x_\mathrm{f}, \mathbf{b}) : \mathrm{f} \neq \mathrm{g}\}$$

By definition, there is no edge from a non-**0** data flow fact to the fact **0** on the exploded super-graph. An edge from the fact **0** to a non-**0** fact means that the non-**0** fact is freshly created [45]. Thus, any summary path in the set S_f^0 does not go through the fact **0**, meaning that the data flow fact is created in a caller of the function f. On the other hand, since a summary path in the set S_f^1 or the set S_f^2 starts with the fact **0**, it means that the non-**0** data flow fact on the summary path must be created in the function f or a callee of the function f. Specifically, since a summary path in the set S_f^1 does not go through the fact **0** in callee functions, the non-**0** data flow fact on the summary path is created in the function f. Similarly, the non-**0** data flow fact on a path from the set S_f^2 must be created in a callee of the function f.

The following lemma states that generating summaries in the sets, S_f^0, S_f^1, and S_f^2, does not miss any summary in the set S_f and, meanwhile, does not repetitively generate a summary in the set S_f.

Lemma 3.1. *$\bigcup_{i \geq 0} S_\mathrm{f}^i = S_\mathrm{f}$ and $\forall i, j \geq 0 : S_\mathrm{f}^i \cap S_\mathrm{f}^j = \emptyset$.*

Proof. This follows the definitions of the sets S_f^0, S_f^1, and S_f^2. □

Next, we study whether such a partition method follows the effectiveness and soundness criteria. The key to the problem is to compute the set $\Omega(S_\mathrm{f}, S_\mathrm{g})$ of dependence relations between a pair of summary sets, S_f^i and S_g^j, given any pair of caller-callee functions, f and g.

Lemma 3.2. *The sets S_f^0, S_f^1, and S_f^2 depend on the set S_g^0.*

Proof. This follows the fact that any summary path in a caller function may go through a callee's summary path and the set S_g^0 is a part of the callee's summaries. □

Lemma 3.3. *The set S_f^2 depends on the sets S_g^1 and S_g^2.*

Proof. By definition, a summary path in the set S_f^2 needs to go through the vertex $(e_\mathrm{g}, \mathbf{0})$. Given the function g, summary paths in both the set S_g^1 and the set S_g^2 start with the vertex $(e_\mathrm{g}, \mathbf{0})$. Thus, the set S_f^2 depends on the sets S_g^1 and S_g^2. □

To demonstrate that the above lemmas do not miss any dependence relations, we establish the following two lemmas.

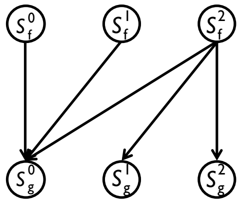


Figure 6: The summary dependence graph for a caller-callee function pair, f and g.

Lemma 3.4. *The set S_{f}^0 does not depend on the sets S_{g}^1 and S_{g}^2.*

Proof. This follows the fact that a non-**0** data flow fact cannot be connected back to the fact **0** [45], but a summary path in the sets S_{g}^1 and S_{g}^2 must start with the fact **0**. □

Lemma 3.5. *The set S_{f}^1 does not depend on the sets S_{g}^1 and S_{g}^2.*

Proof. By definition, a summary path in the set S_{f}^1 does not go through the fact **0** in a callee function. However, a summary path in the sets S_{g}^1 and S_{g}^2 must start with the fact **0**. Thus, the set S_{f}^1 does not depend on the sets S_{g}^1 and S_{g}^2. □

Putting Lemma 3.2 to Lemma 3.5 together, we have the dependence set $\Omega(S_{\mathrm{f}}, S_{\mathrm{g}}) = \{(S_{\mathrm{f}}^0, S_{\mathrm{g}}^0), (S_{\mathrm{f}}^1, S_{\mathrm{g}}^0), (S_{\mathrm{f}}^2, S_{\mathrm{g}}^0), (S_{\mathrm{f}}^2, S_{\mathrm{g}}^1), (S_{\mathrm{f}}^2, S_{\mathrm{g}}^2)\}$, which does not miss any dependence relation between the set S_{f}^i and the set S_{g}^j. Thus, the partition method satisfies the soundness criterion. Meanwhile, $|\Omega(S_{\mathrm{f}}, S_{\mathrm{g}})| = 5 < |\Pi(S_{\mathrm{f}}) \times \Pi(S_{\mathrm{g}})| = 9$. Thus, the effectiveness criterion is satisfied, meaning that the dependence between the summary sets is relaxed and, based on the partition, the parallelism of a bottom-up analysis can be improved.

Figure 6 illustrates the summary dependence graph for a pair of caller-callee functions, f and g. Apparently, based on the graph, when the summaries in the set S_{g}^0 are generated, a bottom-up analysis does not need to wait for summaries in the sets S_{g}^1 and S_{g}^2, but can immediately start generating summaries in the sets S_{f}^0 and S_{f}^1.

3.4 Pipeline Scheduling

As illustrated in Figure 6, given a caller-callee function pair, f and g, we have analyzed the dependence relations between the set S_{f}^i and the set S_{g}^j and shown that the relaxed dependence provides an opportunity to improve the parallelism of a bottom-up analysis. However, we observe that a key problem here is that there are no dependence relations between the sets S_{f}^i and S_{f}^j for a function f, and scheduling the summary-generation tasks for S_{f}^i and S_{f}^j in a random order significantly affects the parallelism.

Figure 7(a) illustrates the worst scheduling method when only one thread is available for each function, respectively. In the scheduling method, the sets S_{f}^0 and S_{g}^0 have the lowest scheduling priority compared to other summary sets. Since all summary sets of the function f depend on the set S_{g}^0, they have to wait for all summary sets of the function g to generate, which is essentially the same as a conventional bottom-up analysis.

Thus, to maximize the parallel performance, given any function g, we need to determine the scheduling priority of the sets S_{g}^0, S_{g}^1, and S_{g}^2. First, as shown in Figure 6, since more summary sets depend on the set S_{g}^0 than the sets S_{g}^1 and S_{g}^2, scheduling the summary-generation task for the set S_{g}^0 in a higher priority will release more tasks for other summary sets.

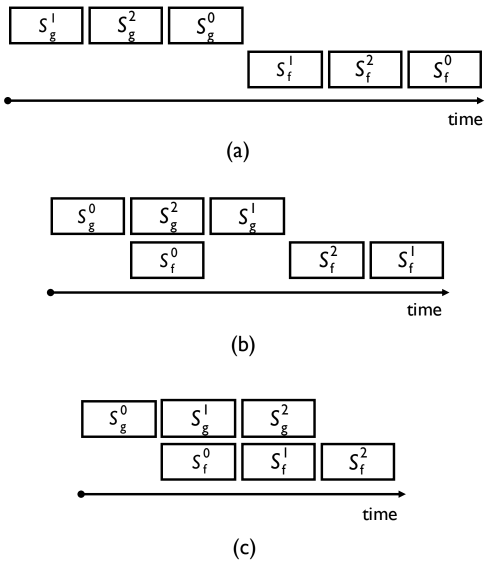


Figure 7: Different scheduling methods when one thread available for each function.

Figures 7(b) and 7(c) illustrate the two possible scheduling methods when for any function g, the set S_{g}^0 is in the highest priority. In Figure 7(b), the set S_{g}^2 has a higher priority than the set S_{g}^1. Since the set S_{f}^2 depends on the sets S_{g}^0, S_{g}^1, and S_{g}^2, it has to wait for all summaries of the function g to generate, leading to a sub-optimal scheduling method. In contrast, Figure 7(c) illustrates the best case where the summary-generation tasks are adequately pipelined.

To conclude, the scheduling priority for any given function g should be $S_{\mathrm{g}}^0 > S_{\mathrm{g}}^1 > S_{\mathrm{g}}^2$, so that the parallelism of a bottom-up analysis can be effectively improved when a limited number of idle threads are available. Such prioritization does not affect the parallelism when there are enough idle threads available.

3.5 ϵ-Bounded Partition and Scheduling

Ideally, the aforementioned partition method evenly partitions a summary set so that the analysis tasks for generating summaries are adequately pipelined, as shown in Figure 7(c). However, in practice, it is usually not the case but works as Figure 8(a), where the sets S_{g}^0 and S_{g}^1 are much larger than other summary sets.

Apparently, if there are extra threads available and we can further partition the summary sets S_{g}^0 and S_{g}^1 into two subsets, the analysis performance then will be improved by generating summaries in the subsets in parallel, just as illustrated in Figure 8(b). Unfortunately, before a bottom-up analysis finishes, we cannot know the actual size of each summary set and, thus, cannot evenly partition a set. As an alternative, what we can do is to approximate an even partition.

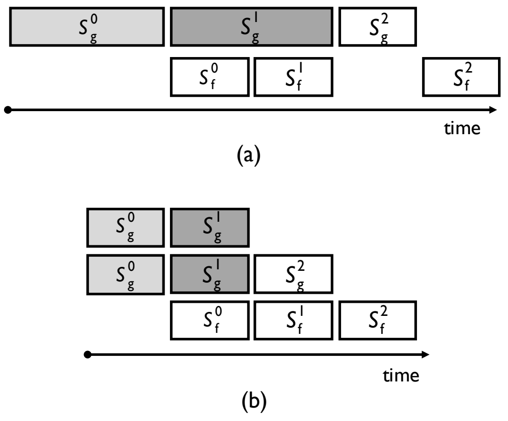


Figure 8: Bounded partition and its scheduling method.

Considering that the analysis task of summary generation is actually to perform a graph traversal from a vertex, we try to further partition a summary set S_f^i based on the number of starting vertices of the graph traversal. To this end, we introduce a client-defined constant ϵ,[3] so that, after the approximately even partition, the graph traversal for generating function summaries in a summary set starts from no more than ϵ vertices.

For example, to generate summaries in the set S_f^0, the analysis needs to traverse the graph G_f from each non-**0** data flow fact at the function entry. Suppose the function f has four non-**0** data flow facts, $\{\mathbf{w}, \mathbf{x}, \mathbf{y}, \mathbf{z}\}$ and $\epsilon = 2$. Then, the set S_f^0 is further partitioned into two subsets $\{(e_f, \mathbf{a}) \rightsquigarrow (x_f, \mathbf{b}) : \mathbf{a} \in \{\mathbf{w}, \mathbf{x}\}\}$ and $\{(e_f, \mathbf{a}) \rightsquigarrow (x_f, \mathbf{b}) : \mathbf{a} \in \{\mathbf{y}, \mathbf{z}\}\}$. After the partition, the graph traversal for both summary sets starts from two vertices.

Similar partition can be performed on the sets S_f^1 and S_f^2 but the following explanation needs to be considered. By definition, it seems difficult to further partition sets S_f^1 and S_f^2 based on the above method, because all summary paths in them start with a single vertex $(e_f, \mathbf{0})$. The key is that, since the fact **0** is a tautology and vertices with the fact **0** are always reachable from each other [45], the graph traversal to generate summaries in the sets S_f^1 and S_f^2 are not necessary to start from the vertex $(e_f, \mathbf{0})$. For instance, since the set S_f^1 contains the summary paths where data flow facts are created in the function f, we can traverse the graph G_f from every vertex that has an immediate predecessor $(s \in L_f, \mathbf{0})$.[4] Similarly, considering that the set S_f^2 contains the summary paths where data flow facts are created in a callee of the function f, we can traverse the graph G_f from every vertex that has an incoming edge from the callees. With multiple starting vertices for the graph traversal, we then can partition the sets S_f^1 and S_f^2 similarly as the set S_f^0.

It is noteworthy that such a bounded partition aims to parallelize the analysis in a single function and, thus, is applicable to both our pipelining approach and the conventional bottom-up approach. Nevertheless, it is particularly useful to improve the pipeline approach as discussed above.

[3]We use $\epsilon = 5$ in our implementation.
[4]Recall that an edge from the fact **0** to a non-**0** data flow fact means the non-**0** fact is freshly created.

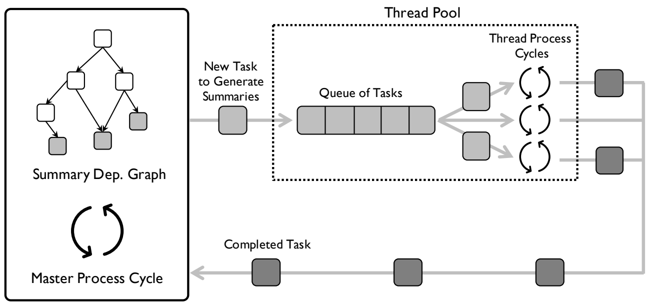


Figure 9: Pipelining bottom-up data flow analysis using a thread pool.

4 IMPLEMENTATION

We have implemented Coyote on top of LLVM[5] to path-sensitively analyze C/C++ programs. This section discusses the implementation details. In the evaluation, for a fair comparison, except for the parallel strategy we study in the paper, all other implementation details are the same in both Coyote and the baseline approaches.

4.1 Parallelization

As illustrated in Figure 9, we implement a thread pool to drive our pipeline parallelization strategy. In the figure, the master process cycle maintains the summary dependence graph for all functions. Each vertex in the graph represents a task to generate certain function summaries. Whenever all of the dependent tasks have been completed, it pushes the current task, referred to as the active task, into a queue and waits for an idle thread to consume it. When a task is completed, the master process cycle is notified so that it can continue to find more active tasks on the dependence graph.

In our implementation, instead of randomly scheduling the tasks in the thread pool, we also seek to design a systematic scheduling method so that we can well-utilize CPU resources. However, it is known that generating an optimal schedule to parallelize the computations in a dependence graph is a variant of precedent-constraint scheduling, which is NP-complete [30]. Therefore, we employ a greedy critical path scheduler [35]. A critical path is the longest remaining path from a vertex to the root vertex on the dependence graph. We then replace the task queue in Figure 9 with a priority queue and prioritize tasks based on the length of critical paths. It is noteworthy that this heuristic scheduling method does not conflict with the pipeline scheduler presented in Section 3.4. The pipeline scheduler prioritizes the analysis tasks in the same function, while the critical-path scheduler only prioritizes the tasks from different functions.

4.2 Taint Analysis

To demonstrate that our approach is applicable to a broad range of data flow analysis, in addition to the null analysis discussed in the paper, we also implement a taint analysis to check two kinds of taint issues. First, we check *relative path traversal*, which allows

[5]LLVM: https://llvm.org/.

an attacker to access files outside of a restricted directory.[6] It is modeled as a path on the exploded super-graph from an external input to a file operation. A typical example is a path from a user input input=gets(...) to a file operation fopen(...). Second, we check *transmission of private resources*, which may leak private data to attackers.[7] It is modeled as a path on the exploded super-graph from sensitive data to I/O operations. A typical example is a path from the password password=getpass(...) to an I/O operation sendmsg(...).

4.3 Pointers and Path-Sensitivity

The null analysis and the taint analysis in Coyote require highly precise pointer information so that they can determine how data flow facts propagate through pointer (load and store) operations. To resolve the pointer relations, we follow the previous work [54] to perform a path-sensitive points-to analysis. The points-to analysis is efficient because it does not exhaustively solve path conditions but records the conditions on the graph edges. When traversing the graph for an analysis, we collect and solve conditions on a path in a demand-driven manner. In Coyote, we use Z3 [13] as the constraint solver to determine path feasibility. According to our experience and many existing works [4, 15, 54, 64], path-sensitivity is a critical factor to make an analysis practical and make the evaluation closer to a real application scenario. For instance, a path-insensitive null analysis reports >90% false positives and, thus, is impractical.

After building the exploded super-graph with the points-to analysis, we simplify the graph via a program slicing procedure, which removes irrelevant edges and vertices, thereby improving the performance of the subsequent null and taint analyses. This simplification process is almost linear to the graph size and, thus, is very fast [46].

As an example, Figure 10(a) is a program where a null pointer is propagated to the variable c through the store and load operations at Line 5 and Line 9. We use the points-to analysis to identify the propagation and build the exploded super-graph as illustrated in Figure 10(b). In this graph, the condition of the propagation y and $\neg y$ are labeled on the edges. Figure 10(c) illustrates the simplified form of the original graph, where unnecessary edges like $(s_{10}, \mathbf{o_b}) \rightsquigarrow (s_{12}, \mathbf{o_b})$ and unnecessary vertices like $(s_8, \mathbf{o_b})$ are removed.

4.4 Soundness

Our implementation of Coyote is soundy [32], meaning that it handles most language features in a sound manner while we also make some well-identified unsound choices following the previous work [4, 10, 54, 58, 64]. Note that Coyote aims to find as many bugs as possible rather than rigorously verifying the correctness of a program. In this context, the unsound choices have limited negative impacts as demonstrated in the previous works. In our implementation, like the previous work [24], we use a flow-insensitive pointer analysis [56] to resolve function pointers. We unroll each cycle twice on both the call graph and the control flow graph [4]. Following the work of Saturn [64], a representative static bug detection tool, we do not model inline assembly and library utilities such as std::vector, std::set, and std::map from the C++ standard template library.

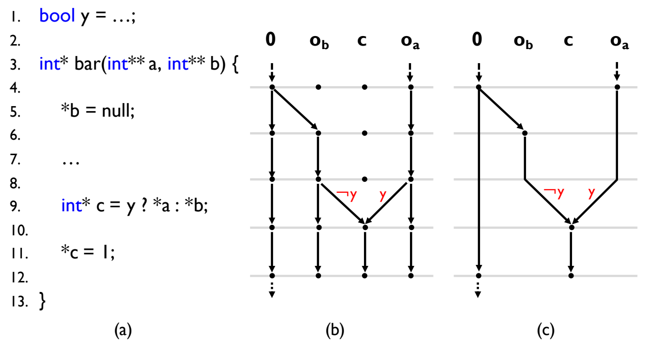

```
1.  bool y = ...;
2.
3.  int* bar(int** a, int** b) {
4.
5.      *b = null;
6.
7.      ...
8.
9.      int* c = y ? *a : *b;
10.
11.     *c = 1;
12.
13. }
```

Figure 10: (a) A code snippet. (b) The exploded super-graph built based on a points-to analysis. (c) The simplified graph. $\mathbf{o_a}$ and $\mathbf{o_b}$ represent the memory object pointed to by a and b, respectively. y and $\neg y$ on the edges are the path conditions.

5 EVALUATION

We now present the experimental setup and the experimental results to demonstrate the effectiveness of our new parallel data flow analysis. We also discuss the factors affecting the evaluation results at the end of this section.

5.1 Experimental Setup

Our goal is to study the scalability of Coyote, a pipeline parallelization strategy for bottom-up data flow analysis. We did this by measuring the CPU utilization rates and the speedup over a conventional parallel implementation. More specifically, a conventional parallel implementation only analyzes functions without calling dependence in parallel, just as illustrated in Figure 1. To precisely measure and study the scalability of our approach, we introduce an artificial throttle that allows us to switch between our pipeline strategy and the conventional parallel strategy. In this manner, we can guarantee that, except for the parallel strategies, all other implementation details discussed in Section 4 are the same for both our approach and the baseline approach. For instance, both approaches accept the same exploded super-graph as the input. Particularly, as discussed in Section 3.5, since the ϵ-bounded partition aims to parallelize the analysis in a single function, it is adopted in both our approach and the baseline approach for a fair comparison. Therefore, the speedup of our approach demonstrated in this section is achieved by the pipeline strategy, i.e., the key contribution of this paper, in isolation. Like the previous work [1], we did not compare our implementation with other tools like Saturn [64] and Calysto [4]. This is because the comparison results will not make any sense due to a lot of different implementation details that may affect the runtime performance.

Our evaluation of Coyote was over the standard SPEC CINT2000 benchmarks,[8] which is commonly used in the literature on static analysis [54, 58]. We also include eight industrial-sized open-source C/C++ projects such as Python, OpenSSL, and MySQL. These real-world subjects are the monthly trending projects on Github that we are able to set up. Table 1 lists the evaluation subjects. The size

[6]CWE-23: https://cwe.mitre.org/data/definitions/23.html.
[7]CWE-402: https://cwe.mitre.org/data/definitions/402.html.

[8]SPEC CINT2000 benchmarks: https://www.spec.org/cpu2000/CINT2000/.

Table 1: Subjects for evaluation.

Origin	ID	Program	Size (KLoC)	# Functions
SPEC CINT2000	1	mcf	2	26
	2	bzip2	3	74
	3	gzip	6	89
	4	parser	8	324
	5	vpr	11	272
	6	crafty	13	108
	7	twolf	18	191
	8	eon	22	3,367
	9	gap	36	843
	10	vortex	49	923
	11	perlbmk	73	1,069
	12	gcc	135	2,220
Open Source	13	bftpd	5	260
	14	shadowsocks	32	574
	15	webassembly	75	7,842
	16	redis	101	1,527
	17	python	434	3,619
	18	icu	537	27,046
	19	openssl	791	11,759
	20	mysql	2,030	79,263
			Total 4,381	**Avg.** 7,070

of these subjects is more than four million lines of code in total, ranging from a few thousand to two million lines of code. The number of functions of these subjects ranges from tens to nearly eighty thousand functions, with about seven thousand on average.

We ran our experiments on a server with eighty "Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz" processors and 256GB of memory running Ubuntu-16.04. We set our initial number of threads to be twenty and added twenty for every subsequent run until the maximum number of available processors, i.e., eighty. All the experiments were run with the resource limitation of twelve hours.

5.2 Study of the Null Analysis

We first present the experimental results of the null analysis in detail, followed by a brief discussion on the taint analysis in the next subsection.

5.2.1 Speedup. Table 2 lists the comparison results of the conventional parallel mechanism (Conv) and our pipeline strategy (Pipeline) for the bottom-up program analysis. Each row of the table represents the results of a benchmark program, including the time cost in seconds and the speedup for these two kinds of parallel mechanisms. The speedup is calculated as the ratio of the time taken by Coyote to that of the conventional parallel approach with the same number of threads.

We observe that the speedup achieved with 20 threads is 1.5× on average. However, as the number of threads is increased to 80, the observed speedup also increases, up to 3× faster. Using several typical examples, Figure 11 illustrates the relation between the number of threads and the speedup. The growing curves show that the speedup increases with the growth of the number of threads, demonstrating that we can always achieve speedup and have higher parallelism than the conventional parallel approach.

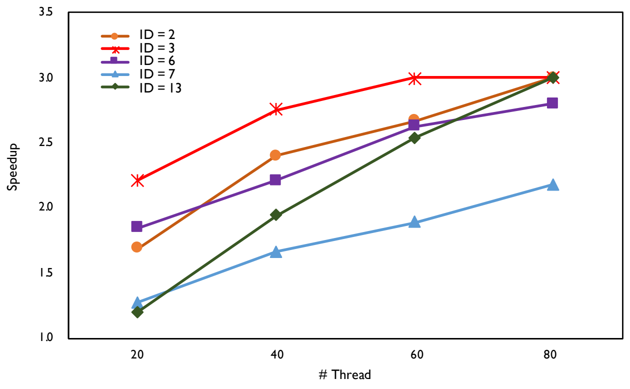


Figure 11: Speedup vs. The number of threads.

It is noteworthy that such 2×-3× speedup is significant enough to make many overly lengthy analyses useful in practice. For example, originally, it takes more than 10 hours to analyze MySQL (ID = 20, Size = 2 MLoC, typical size in industry). The time cost cannot satisfy the industrial requirement of finishing analysis in 5 to 10 hours [35]. With the pipeline strategy, it saves more than 6 hours, making the bug finding task acceptable in the industrial setting.

5.2.2 CPU Utilization Rate. The speedup over the conventional parallel design is due to the higher parallelism achieved by the pipeline strategy. To quantify this effect, we profile the CPU utilization rates for both the conventional parallel design and the pipeline method. Figure 12 demonstrates the CPU utilization rates against the elapsed running time. Due to the page limit, we only show several typical ones for some of the programs running with 80 threads. In the figure, the solid line represents the CPU utilization rate of our pipeline method while the dashed line represents that of the conventional parallel design.

We can observe that, for each project, in the initial phase of the analysis, the CPU utilization rates for both parallel designs are similar, almost occupying all available CPUs. This is because the call graph of a program is usually a tree-like data structure. In the bottom half of the call graph, it usually has enough independent functions that we can analyze in parallel. Thus, both parallel designs can sufficiently utilize the CPUs.

Our pipeline strategy unleashes its power in the remaining part of the analysis, where it apparently has much higher CPU utilization rates, thus finishing the analysis much earlier. This is because the top half of a call graph is much denser, where there are more calling relations than the bottom half. Since the conventional parallel design cannot analyze functions with calling relations in parallel, it cannot sufficiently utilize the CPUs. In contrast, our approach splits the analysis of a function into multiple parts and allows us to analyze functions with calling relations in parallel, thus being able to utilize more CPUs.

5.3 Study of the Taint Analysis

In order to demonstrate that our approach is generalizable to other analyses, we also conducted an experiment to see whether the pipeline approach can improve the scalability of taint analysis.

Table 2: Running time (seconds) and the speedup over the conventional parallel design of bottom-up analysis.

ID	# Thread = 20			# Thread = 40			# Thread = 60			# Thread = 80		
	Conv	Pipeline	Speedup	Conv	Pipeline	Speedup	Conv	Pipeline	Speedup	Conv	Pipeline	**Speedup**
1	60	28	2.1×	60	24	2.5×	60	20	3.0×	60	20	**3.0×**
2	108	64	1.7×	96	40	2.4×	96	36	2.7×	96	32	**3.0×**
3	168	76	2.2×	168	61	2.8×	168	56	3.0×	168	56	**3.0×**
4	252	215	1.2×	168	120	1.4×	132	92	1.4×	132	72	**1.8×**
5	264	192	1.4×	180	116	1.6×	156	88	1.8×	144	76	**1.9×**
6	192	104	1.8×	168	76	2.2×	168	64	2.6×	168	60	**2.8×**
7	168	132	1.3×	133	80	1.7×	121	64	1.9×	122	56	**2.2×**
8	2568	2148	1.2×	1620	1192	1.4×	1296	865	1.5×	1128	708	**1.6×**
9	1728	860	2.0×	1524	648	2.4×	1500	576	2.6×	1476	545	**2.7×**
10	843	648	1.3×	698	374	1.9×	674	280	2.4×	662	252	**2.6×**
11	1530	913	1.7×	1325	604	2.2×	1232	528	2.3×	1217	500	**2.4×**
12	1978	1573	1.3×	1486	926	1.6×	1306	729	1.8×	1235	613	**2.0×**
13	156	109	1.4×	132	68	1.9×	132	52	2.5×	132	44	**3.0×**
14	876	468	1.9×	780	340	2.3×	768	296	2.6×	768	288	**2.7×**
15	2940	1990	1.5×	2292	1248	1.8×	2076	1012	2.1×	1980	908	**2.2×**
16	1332	1060	1.3×	984	628	1.6×	900	488	1.8×	864	416	**2.1×**
17	5162	3022	1.7×	4276	2036	2.1×	4035	1738	2.3×	3895	1605	**2.4×**
18	7.8hr	5.5hr	1.4×	5.8hr	3.4hr	1.7×	5.2hr	2.6hr	2.0×	4.9hr	2.3hr	**2.1×**
19	2.8hr	2.2hr	1.2×	1.9hr	1.2hr	1.6×	1.7hr	0.9hr	1.9×	1.6hr	0.8hr	**2.0×**
20	Time Out	9.6hr	-	Time Out	7.8hr	-	Time Out	6.4hr	-	11.8hr	5.6hr	**2.1×**

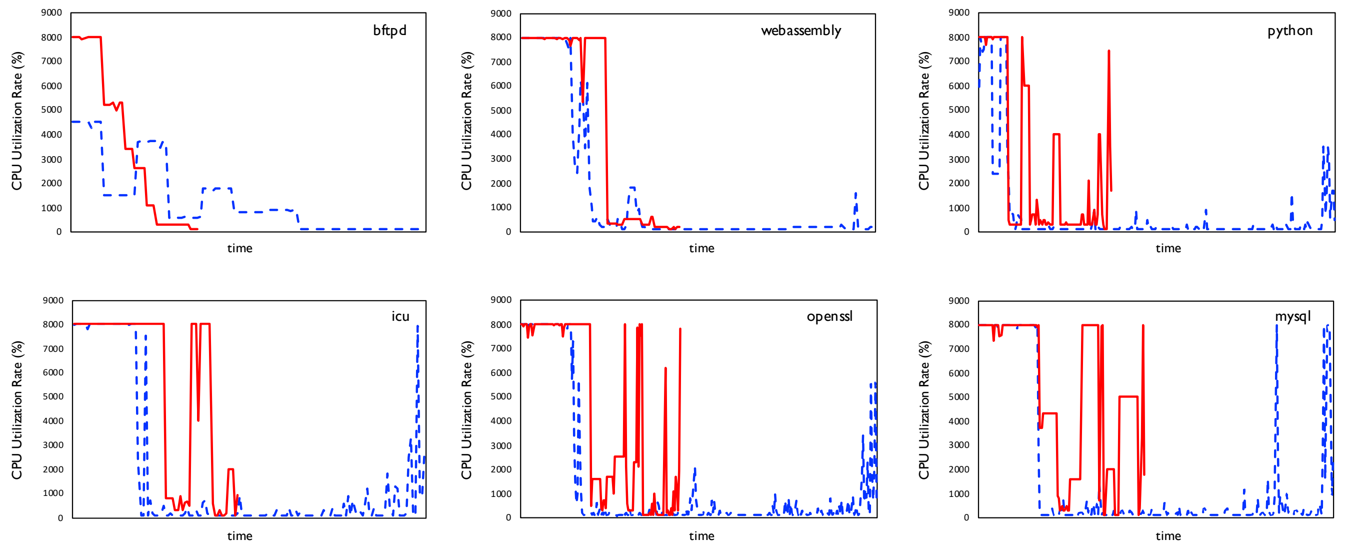


Figure 12: CPU utilization rate vs. The elapsed time. The solid lines represent the CPU utilization rate of our pipeline method while the dashed lines represent that of the conventional parallel design.

Since the result of taint analysis are quite similar to that of the null analysis, we briefly summarize the experimental results in Table 3, where the results of our largest benchmark program, MySQL, are presented. The results demonstrate that, with the increase of the number of available threads, the speedup of our approach over the conventional approach also grows to >2× in analyzing both the *relative path traversal* (RPT) bug or the *transmission of private resources* (TPR) bug.

5.4 Discussion

There are two main factors affecting the evaluation results: the density of the call graph and the number of available threads.

As discussed above, when the call graph is very sparse, the advantage of our approach is not very obvious. For instance, if functions are all independent on each other, all functions can be run in parallel. Thus, both approaches can always sufficiently utilize the available threads and, thus, have similar time efficiency. In practice, as demonstrated in our evaluation, the call graph is usually tree-like.

Table 3: Results of the taint analysis on MySQL.

Taint Issues	# Thread = 20			# Thread = 40			# Thread = 60			# Thread = 80		
	Conv	Pipeline	Speedup	Conv	Pipeline	Speedup	Conv	Pipeline	Speedup	Conv	Pipeline	**Speedup**
RPT	Time Out	10.2hr	-	Time Out	8.7hr	-	Time Out	7.1hr	-	10.9hr	4.7hr	**2.3×**
TPR	9.3hr	6.6hr	1.4×	8.1hr	5.0hr	1.6×	7.4hr	3.9hr	1.9×	6.1hr	2.8hr	**2.2×**

Thus, our approach can present its power in the second half of the analysis and achieves up to 3× speedup in practice.

The number of threads is also a key factor affecting the observed speedup of our approach. For instance, if we only have one thread available, although our approach can provide more independent tasks, these tasks cannot be run in parallel. Thus, both of our approach and the conventional one will emit similar results. As illustrated by the evaluation, our approach can work better when we have more available threads. In the cloud era, we can expect that we have unlimited CPU resources and, thus, can expect more benefits from our approach in practice.

6 RELATED WORK

Parallel and distributed algorithms for data flow analysis is an active area of research. In this section, we survey existing parallel or distributed techniques and compare them with Coyote.

In order to utilize the modular structure of a program to parallelize the analyses in different functions, developers usually implement a data flow analysis in a top-down fashion or a bottom-up manner. Albarghouthi et al. [1] presented a generic framework to distribute top-down algorithms using a map-reduce strategy. Parallel worklist approaches, a kind of top-down analysis, also can address the IFDS/IDE problems. They operate by processing the elements on an analysis worklist in parallel [14, 21, 48]. These approaches are different from ours because this paper focuses on bottom-up analysis. In our opinion, the top-down approach and the bottom-up approach are two separate schools of methodologies to implement program analysis. Bottom-up approaches analyze each function only once and generate summaries reusable at all calling contexts. Top-down approaches generate summaries that are specific to individual calling contexts and, thus, may need to repeat analyzing a function. For analyses that need high precision like path-sensitivity, repetitively analyzing a function is costly. Thus, we may expect better performance from bottom-up analysis when high precision is required.

Compared to top-down analysis, bottom-up analysis has been traditionally easier to parallelize. Existing static analyses, such as Saturn [64], Calysto [4], Pinpoint [54], and Infer [8], have utilized the function-level parallelization to improve their scalability. However, none of them presented any techniques to further improve its parallelism. McPeak et al. [35] pointed out that the CPU utilization rate may drop in the dense part of the call graph where the parallelism is significantly limited by the calling dependence. Although they presented an optimized scheduling method to mitigate the performance issue, the calling dependence was not relaxed and the function-level parallelism was not improved. We believe that their scheduling method is complementary to Coyote and their combination has the potential for the greater scalability.

In contrast to top-down and bottom-up approaches, partition-based approaches [6, 12, 17, 22, 26, 29, 33, 38] do not utilize the modular structure of a program but partition the state space and distribute the state-space search to several threads or processors. Another category of data flow analyses (e.g., [2, 7, 25]) are modeled as Datalog queries rather than the graph reachability queries in the IFDS/IDE framework. They can benefit from parallel Datalog engines to improve the scalability [20, 27, 28, 34, 51–53, 61, 62, 66].

Recently, some other parallel techniques have been proposed. Many of them focus on pointer analysis [18, 31, 37, 41, 44, 57] rather than general data flow analysis. Mendez-Lojo et al. [36] proposed a GPU-based implementation for inclusion-based pointer analysis. EigenCFA [43] is a GPU-based flow analysis for higher-order programs. Graspan [60] and Grapple [68] turn sophisticated code analysis into big data analytics. They utilize recent advances on solid-state disks to parallelize and scale program analysis. These techniques are not designed for compositional data flow analysis and, thus, are different from our approach.

In addition to automatic techniques, Ball et al. [5] used manually created harnesses to specify independent device driver entry points so that an embarrassingly parallel workload can be created.

7 CONCLUSION

We have presented Coyote, a pipeline parallelization strategy that enables to perform bottom-up data flow analysis in a faster way. The pipeline strategy relaxes the calling dependence, which conventionally limits the parallelism of bottom-up analysis. The evaluation of our approach demonstrates higher CPU utilization rates and significant speedup over a conventional parallel design. In the multi-core era, we believe that improving the parallelism is an important approach to scaling static program analysis.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their insightful comments. This work is partially funded by an MSRA grant, as well as Hong Kong GRF16230716, GRF16206517, ITS/215/16FP, and ITS/440/18FP grants.

REFERENCES

[1] Aws Albarghouthi, Rahul Kumar, Aditya V Nori, and Sriram K Rajamani. 2012. Parallelizing top-down interprocedural analyses. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, 217–228.

[2] Nicholas Allen, Padmanabhan Krishnan, and Bernhard Scholz. 2015. Combining type-analysis with points-to analysis for analyzing Java library source-code. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis (SOAP '15)*. ACM, 13–18.

[3] 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 (PLDI '14)*. ACM, 259–269.

[4] Domagoj Babic and Alan J. Hu. 2008. Calysto: Scalable and precise extended static checking. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. IEEE, 211–220.

[5] Thomas Ball, Vladimir Levin, and Sriram K Rajamani. 2011. A decade of software model checking with SLAM. *Commun. ACM* 54, 7 (2011), 68–76.

[6] Jiri Barnat, Lubos Brim, and Jitka Střibrná. 2001. Distributed LTL model-checking in SPIN. In *International SPIN Workshop on Model Checking of Software*. Springer, 200–216.

[7] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, 243–262.

[8] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional shape analysis by means of bi-abduction. *J. ACM* 58, 6 (2011), 26:1–26:66.

[9] Sagar Chaki, Edmund M Clarke, Alex Groce, Somesh Jha, and Helmut Veith. 2004. Modular verification of software components in C. *IEEE Transactions on Software Engineering* 30, 6 (2004), 388–402.

[10] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. 2007. Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, 480–491.

[11] Chia Yuan Cho, Vijay D'Silva, and Dawn Song. 2013. BLITZ: Compositional bounded model checking for real-world programs. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE '13)*. IEEE, 136–146.

[12] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. 2010. Cloud9: A software testing service. *ACM SIGOPS Operating Systems Review* 43, 4 (2010), 5–10.

[13] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[14] Kyle Dewey, Vineeth Kashyap, and Ben Hardekopf. 2015. A parallel abstract interpreter for JavaScript. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO '15)*. IEEE, 34–45.

[15] Isil Dillig, Thomas Dillig, and Alex Aiken. 2008. Sound, complete and scalable path-sensitive analysis. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, 270–280.

[16] Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. 2011. Precise and compact modular procedure summaries for heap manipulating programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, 567–577.

[17] Matthew B Dwyer, Sebastian Elbaum, Suzette Person, and Rahul Purandare. 2007. Parallel randomized state-space search. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE, 3–12.

[18] Marcus Edvinsson, Jonas Lundberg, and Welf Löwe. 2011. Parallel points-to analysis for multi-core machines. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*. ACM, 45–54.

[19] Stephen J Fink, Eran Yahav, Nurit Dor, G Ramalingam, and Emmanuel Geay. 2008. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 17, 2 (2008), 9.

[20] Sumit Ganguly, Avi Silberschatz, and Shalom Tsur. 1990. A Framework for the Parallel Processing of Datalog Queries. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data (SIGMOD '90)*. ACM, 143–152.

[21] Diego Garbervetsky, Edgardo Zoppi, and Benjamin Livshits. 2017. Toward full elasticity in distributed static analysis: the case of callgraph analysis. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE '17)*. ACM, 442–453.

[22] Orna Grumberg, Tamir Heyman, Nili Ifergan, and Assaf Schuster. 2005. Achieving speedups in distributed symbolic reachability analysis through asynchronous computation. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Springer, 129–145.

[23] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. 2011. Saving the world wide web from vulnerable JavaScript. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11)*. ACM, 177–187.

[24] Ben Hardekopf and Calvin Lin. 2011. Flow-sensitive pointer analysis for millions of lines of code. In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*. IEEE, 289–298.

[25] Behnaz Hassanshahi, Raghavendra Kagalavadi Ramesh, Padmanabhan Krishnan, Bernhard Scholz, and Yi Lu. 2017. An efficient tunable selective points-to analysis for large codebases. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis (SOAP '17)*. ACM, 13–18.

[26] Gerard J Holzmann and Dragan Bosnacki. 2007. The design of a multicore extension of the SPIN model checker. *IEEE Transactions on Software Engineering* 33, 10 (2007), 659–674.

[27] G. Hulin. 1989. Parallel Processing of Recursive Queries in Distributed Architectures. In *Proceedings of the 15th International Conference on Very Large Data Bases (VLDB '89)*. Morgan Kaufmann Publishers Inc., 87–96.

[28] Herbert Jordan, Pavle Subotić, David Zhao, and Bernhard Scholz. 2019. A specialized B-tree for concurrent datalog evaluation. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. ACM, 327–339.

[29] Yong-fong Lee and Barbara G Ryder. 1992. A comprehensive approach to parallel data flow analysis. In *Proceedings of the 6th International Conference on Supercomputing*. ACM, 236–247.

[30] Jan Karel Lenstra and AHG Rinnooy Kan. 1978. Complexity of scheduling under precedence constraints. *Operations Research* 26, 1 (1978), 22–35.

[31] Bozhen Liu, Jeff Huang, and Lawrence Rauchwerger. 2019. Rethinking Incremental and Parallel Pointer Analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 41, 1 (2019), 6.

[32] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundiness: a manifesto. *Commun. ACM* 58, 2 (2015), 44–46.

[33] Nuno P Lopes and Andrey Rybalchenko. 2011. Distributed and predictable software model checking. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 340–355.

[34] Carlos Alberto Martínez-Angeles, Inês Dutra, Vítor Santos Costa, and Jorge Buenabad-Chávez. 2013. A datalog engine for gpus. In *Declarative Programming and Knowledge Management*. Springer, 152–168.

[35] Scott McPeak, Charles-Henri Gros, and Murali Krishna Ramanathan. 2013. Scalable and incremental software bug detection. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '13)*. ACM, 554–564.

[36] Mario Mendez-Lojo, Martin Burtscher, and Keshav Pingali. 2012. A GPU implementation of inclusion-based points-to analysis. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. ACM, 107–116.

[37] Mario Méndez-Lojo, Augustine Mathew, and Keshav Pingali. 2010. Parallel inclusion-based points-to analysis. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, 428–443.

[38] David Monniaux. 2005. The parallel implementation of the Astrée static analyzer. In *Asian Symposium on Programming Languages and Systems*. Springer, 86–96.

[39] Nomair A Naeem and Ondrej Lhotak. 2008. Typestate-like analysis of multiple interacting objects. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA '08)*. ACM, 347–366.

[40] Nomair A Naeem and Ondrej Lhoták. 2009. Efficient alias set analysis using SSA form. In *Proceedings of the 2009 International Symposium on Memory Management (ISMM '09)*. ACM, 79–88.

[41] Vaivaswatha Nagaraj and R Govindarajan. 2013. Parallel flow-sensitive pointer analysis by graph-rewriting. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 19–28.

[42] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. 2013. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security '13)*. USENIX Association, 543–558.

[43] Tarun Prabhu, Shreyas Ramalingam, Matthew Might, and Mary Hall. 2011. EigenCFA: Accelerating flow analysis with GPUs. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, 511–522.

[44] Sandeep Putta and Rupesh Nasre. 2012. Parallel replication-based points-to analysis. In *International Conference on Compiler Construction (CC '12)*. Springer, 61–80.

[45] 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 (POPL '95)*. ACM, 49–61.

[46] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. 1994. Speeding up slicing. In *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE '94)*. ACM, 11–20.

[47] Noam Rinetzky, Mooly Sagiv, and Eran Yahav. 2005. Interprocedural shape analysis for cutpoint-free programs. In *International Static Analysis Symposium*. Springer, 284–302.

[48] Jonathan Rodriguez and Ondřej Lhoták. 2011. Actor-based parallel dataflow analysis. In *International Conference on Compiler Construction (CC '11)*. Springer, 179–197.

[49] Atanas Rountev, Mariana Sharp, and Guoqing Xu. 2008. IDE dataflow analysis in the presence of large object-oriented libraries. In *International Conference on Compiler Construction (CC '08)*. Springer, 53–68.

[50] Mooly Sagiv, Thomas Reps, and Susan Horwitz. 1996. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science* 167, 1 (1996), 131–170.

[51] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On fast large-scale program analysis in datalog. In *International Conference on Compiler Construction (CC '16)*. ACM, 196–206.
[52] Jürgen Seib and Georg Lausen. 1991. Parallelizing Datalog programs by generalized pivoting. In *Proceedings of the tenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. ACM, 241–251.
[53] Marianne Shaw, Paraschos Koutris, Bill Howe, and Dan Suciu. 2012. Optimizing large-scale Semi-Naïve datalog evaluation in hadoop. In *International Datalog 2.0 Workshop*. Springer, 165–176.
[54] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: Fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '18)*. ACM, 693–706.
[55] Sharon Shoham, Eran Yahav, Stephen J Fink, and Marco Pistoia. 2008. Static specification mining using automata-based abstractions. *IEEE Transactions on Software Engineering* 34, 5 (2008), 651–666.
[56] Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 32–41.
[57] Yu Su, Ding Ye, and Jingling Xue. 2014. Parallel pointer analysis with CFL-reachability. In *2014 43rd International Conference on Parallel Processing*. IEEE, 451–460.
[58] Yulei Sui, Ding Ye, and Jingling Xue. 2014. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering* 40, 2 (2014), 107–122.
[59] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. 2013. Andromeda: Accurate and scalable security analysis of web applications. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 210–225.
[60] 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. *ACM SIGOPS Operating Systems Review* 51, 2 (2017), 389–404.
[61] Ouri Wolfson and Aya Ozeri. 1990. A New Paradigm for Parallel and Distributed Rule-processing. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data (SIGMOD '90)*. ACM, 133–142.
[62] Ouri Wolfson and Avi Silberschatz. 1988. Distributed Processing of Logic Programs. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data (SIGMOD '88)*. ACM, 329–336.
[63] Yichen Xie and Alex Aiken. 2005. Context- and path-sensitive memory leak detection. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE '05)*. ACM, 115–125.
[64] Yichen Xie and Alex Aiken. 2005. Scalable error detection using Boolean satisfiability. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*. ACM, 351–363.
[65] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter O'Hearn. 2008. Scalable shape analysis for systems code. In *International Conference on Computer Aided Verification*. Springer, 385–398.
[66] Mohan Yang, Alexander Shkapsky, and Carlo Zaniolo. 2017. Scaling up the performance of more powerful Datalog systems on multicore machines. *The VLDB Journal - The International Journal on Very Large Data Bases* 26, 2 (2017), 229–248.
[67] Greta Yorsh, Eran Yahav, and Satish Chandra. 2008. Generating precise and concise procedure summaries. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. ACM, 221–234.
[68] 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 (EuroSys '19)*. ACM, 38.