Towards Efficient Large-scale Interprocedural Program Static Analysis on Distributed Data-Parallel Computation

— Extended Version —

Rong Gu, Zhiqiang Zuo*, Xi Jiang, Han Yin, Zhaokang Wang,

Linzhang Wang, Xuandong Li, and Yihua Huang*
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China
Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China

Abstract—Static program analysis has been widely applied along the whole process of the program development for bug detection, code optimization, testing, etc. Although researchers have made significant work in static program analysis, it is still challenging to perform sophisticated interprocedural analysis on large-scale modern software. The underlying reason is that interprocedural analysis for large-scale modern software is highly computation- and memory-intensive, leading to poor efficiency and scalability. In this paper, we introduce an efficient distributed and scalable solution for sophisticated static analysis. Specifically, we propose a data-parallel algorithm and a *join-process-filter* computation model for the CFL-reachability based interprocedural analysis. Based on that, an efficient distributed static analysis engine called BigSpa is developed, which is composed of an offline batch static program analysis system and an online incremental static program analysis system. The BigSpa system has high generality and can support all kinds of static analysis tasks that can be expressed as CFL reachability problems. The performance of BigSpa is evaluated on real-world large-scale software datasets. Our experiments show that the offline batch system can exceed an order of magnitude compared with the most advanced analysis tools available on performance, and for incremental analysis with small batch updates on the same data sets, the online analysis system can achieve near real-time response, which is very fast and flexible.

Index Terms—Interprocedural static analysis, distributed systems, data-parallel computation.

1 Introduction

In the entire process of software development, interprocedural static program analyses are widely used for tasks such as bug detection, code optimization, testing, debugging and so on.

Motivation: Nowadays, people apply the simple pattern-based or intraprocedural analysis to uncover various software problems in practice ([1], [2], [3]). However, due to the lack of considering adequately rich/complete semantics information, such simple analyses suffer from severely low accuracy (30%-100% false positive rate [4]), leading to dead poor applicability[5]. For the sake of precise and thus useful analysis results, sophisticated interprocedural analyses are essential, e.g., interprocedural context-, path-, flow-, field-sensitive analysis. Different from the imprecise but efficient intraprocedural analysis, interprocedural analyses take into account more semantics information, resulting in high analysis complexity. For instance, in a context-sensitive interprocedural analysis, the number of calling contexts can easily exceed millions[6], making the analysis both computation- and memory-intensive. In addition, as the sizes of modern software grow, performing precise analysis on real-world modern software systems like Linux kernel, becomes more and more challenging. It is reported that in a context-sensitive interprocedural alias analysis for Linux kernel (more than 16M lines of code), there are more than 1 billion edges generated. None of the state-of-the-art analysis tools can accomplish the analysis task due to either analysis time or memory usage issues[7]. The poor scalability greatly hinders the interprocedural analysis being more widely adopted in industry.

The authors are with State Key Laboratory for Novel Software Technology, and Department of Computer Science and Technology, Nanjing University, China. Zhiqiang Zuo and Yihua Huang are corresponding authors with equal contribution.
E-mail:{gurong, zqzuo}@nju.edu.cn,{xijiang, hanyin, zhaokangwang}@smail.nju.edu.cn,{lzwang, lxd, yhuang}@nju.edu.cn.

From our point of view, the reason of poor scalability is two-fold. First, the majority of existing interprocedural analysis systems are usually based on sequential algorithms. They lack the support for highly efficient parallel and distributed computing. Second, most state-of-the-art algorithms are purely memory-based. The limited size of available memory severely impedes the applicability of static analyses in large-scale scenarios.

Researchers have been endeavouring to improve the analysis scalability by proposing approximations, abstractions, or various heuristics. However, approximations usually trade off analysis capability for scalability, rendering analysis less precise and useful. Even worse, the analysis with lower accuracy sometimes still failed to analyze large-scale systems code. In short, it is still too challenging to efficiently perform interprocedural analysis on large-scale codebases.

Approach: Previous work[7] introduces the idea of turning Big Code analysis into Big Data analytics. Unfortunately, as for the sophisticated and large-scale analysis workloads, it still severely suffers from slow convergence and/or execution failure due to the limited scalability of one single machine. This work offers a more scalable solution by leveraging the distributed computing facilities. We follow the direction of "Big Data" solution, employ it under the distributed setting, and develop a highly scalable system, thus making a step forward in large-scale static analysis.

Pioneered by [8], [9], we formulate sophisticated interprocedural analyses as grammar-guided graph reachability, particularly the context-free-language (CFL) reachability problems [10]. Pointer/alias analysis and dataflow analysis are two typical examples. For instance, in a pointer analysis, if a variable is reachable from an object instantiated via a malloc by following certain path in a directed program graph representation, the variable may point to that object. Similarly, for a dataflow analysis that uncovers all the NULL pointer variables in a program, a path from NULL to a variable indicates the NULL value of the variable. Based on this insight, we turn programs into graphs and treat the analyses as graph processing problems.

However, none of the existing big graph processing systems (e.g., GraphLab[11], GraphX[12], GraphChi[13], RStream[14]) supports this type of tasks well as explained in [7] due to the unique characteristic the CFL reachability possesses: dynamically computing reachability by repeatedly adding transitive edges into the graph.

Contributions: In this work, we make the following contributions:

- We devise a data-parallel algorithm and a *join-process-filter* computation model for CFL-reachability based interprocedural analysis. Based on that, a distributed offline batch static program analysis system is implemented. To further improve performance, we propose both algorithm- and system-level optimizations, including computation closure to reduce iteration rounds, finer-grained partitioning for load balancing, and pre-shuffle to save network traffics. We also design a data structure with compression effect and manage to reduce the memory usage to one-third of the original.
- Further, to deal with mini-batch code updating scenarios, an online incremental static program analysis system supporting small batches of code update is designed and implemented. A triangle counting method together with an improved data structure for online analysis is proposed to support the continuous updating of the graph topology. Considering the occurrence of large-scale computation triggered by a small batch update, we design a mechanism for the computation scale prediction and automatic computation mode switching.
- Based on the big data processing platform Spark, distributed file system HDFS and distributed in-memory database Redis, we implement a system and framework called BigSpa, integrating computation, storage and query for large-scale static program analysis. The system has high generality and can support all kinds of static analysis tasks that can be expressed as CFL reachability problems. We have made BigSpa publicly available at https://github.com/PasaLab/BigSpa.
- We empirically validate the BigSpa's performance with extensive experiments over real-world large-scale software datasets. The experimental results show that compared with existing state-of-the-art analysis tools, our offline batch static program analysis system is one or more orders of magnitude faster. Also, for small batch updates, the online analysis system can make a near real-time response in seconds, which is very fast and flexible.

The rest of this paper is organized as follows. Section 2 introduces the background of this paper. Section 3 revisits the static analysis problem from the "Big Data" angle and proposes a scalable data-parallel solution. Section 4 introduces the design and optimization details of the offline batch program static analysis system. Section 5 further extends the previous section and designs an online incremental system. Section 6 introduces the design and usage details of the entire framework. The performance evaluation results are presented and analyzed in Section 7. We discuss related work in Section 8, and concludes the paper in Section 9 finally.

2 Background

This section provides the necessary background about CFL-reachability based static analysis. We demonstrate how two typical analysis examples (i.e., pointer/alias analysis and dataflow analysis) are formulated as CFL reachability problems, and introduce the traditional worklist algorithm for CFL reachability.

2.1 CFL Reachability

CFL-reachability is initially proposed by Yannakakis[15] for Datalog query evaluation. Later, a large body of program analysis problems [8], [9], [16] are formulated as CFL reachability instances.

Informally, the CFL reachability computation is often guided by a context-free language $\mathcal{L}$ over an alphabet Σ. Given a graph $\mathcal{G}$, whose edges are labelled with members of Σ. A vertex in $\mathcal{G}$ is $\mathcal{L}$-reachable from another vertex if and only if there exists a path between them, the word formed by concatenating labels along which is a member of $\mathcal{L}$. The (all-pairs) CFL reachability problem is to determine all pairs of vertices v, w such that w is $\mathcal{L}$-reachable from v.

2.2 Two Examples

We give two typical analysis examples as CFL reachability problems in the following.

Alias Analysis: Two variables are aliased if they point to the same memory location. Alias analysis is used to determine if two variables are aliases. Alias analysis can be formulated as CFL-reachability under the program expression graph representation of programs [16]. The analysis we implement is *flow-insensitive* in the sense that we do not care about the program control flow.

Based on the formulation, a program expression graph is constructed where each pointer expression (reference variable x, dereference expression $*x$, and address-of expression $\&x$) corresponds to a graph vertex. Five kinds of three-address statements listed below are considered to add respective edges into the graph.

Type	***Stmt***	***Edge***	
memory allocation	$x = malloc()$	$x \overset{m}{\leftarrow} A$	(1)
assignment	$x = y$	$x \overset{a}{\leftarrow} y$	(2)
store	$*x = y$	$*x \overset{a}{\leftarrow} y$	(3)
load	$x = *y$	$x \overset{a}{\leftarrow} *y$	(4)
address-of	$x = \&y$	$x \overset{a}{\leftarrow} \&y$	(5)

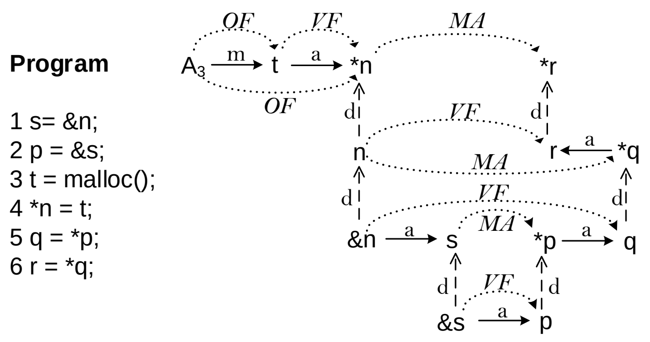


Fig. 1: A program and its expression graph: solid, horizontal edges represent assignments (a- and m- edges); dashed, vertical edges represent dereferences (d-edge); dotted, horizontal edges represent transitive edges labelled with non-terminals. A_3 indicates the allocation site at Line 3.

As for a memory allocation, we add the edge from A to x labeled by m. All other four types of statements would trigger the addition of a edge, accordingly. Moreover, we add *dereference edge (d)* from each pointer variable x to $*x$ and from $\&x$ to x.

Type	***Edge***	
dereference	$*x \overset{d}{\dashleftarrow} x$	(6)
	$x \overset{d}{\dashleftarrow} \&x$	(7)

Given a program expression graph, the language adopted in the alias analysis is as follows:

$$Object\ flow \quad OF ::= m\ VF \tag{8}$$

$$Value\ flow \quad VF ::= (a\ MA? \mid MA?\ a)^* \tag{9}$$

$$Memory\ alias \quad MA ::= \overline{d}\ VF\ d \tag{10}$$

A non-terminal OF represents the points-to relation. One vertex x in the expression graph which is OF-reachable from o (i.e., there is a path from o to x, along which the labels form a word of OF), means that a variable x points to an object o during execution. The definition of OF is straightforward: it must start with an alloc (m) edge, followed by a VF path that propagates the object address to a variable. A VF path is either a sequence of simple assignment (a) edges or a mix of assignments edges and *MA* (memory alias) paths. An *MA* path is represented by $\overline{d}\ VF\ d$. Each edge has an inverse edge with a "bar" label. If $o \xrightarrow{d} x$ occurs in the graph, $x \xrightarrow{\overline{d}} o$ also exists. $\overline{d}$ represents the inverse of a dereference and is essentially equivalent to an address-of. $\overline{d}\ VF\ d$ represents the fact that if we take the address of a variable x, propagate the address through a VF path to another variable y, and then do a dereference on y, the result is the same as the value in x.

Figure 1 shows a simple program and its expression graph. Solid and dashed edges are original edges in the graph and they are labeled m, a, or d, respectively. Dotted edges are transitive edges added by BigSpa into the graph, as discussed shortly.

In Figure 1, t points to A_3, since the m edge between them forms an OF path. There is a VF path from $\&s$ to p, which enables an MA path from s to $*p$ due to the balanced parentheses d and $\overline{d}$. This path then induces an additional VF path from $\&n$ to q, which, in turn, contributes to the forming of the VF path from n to r, making $*n$ and $*r$ memory aliases. The dotted edges in Figure 1 show these transitive edges.

Dataflow Analysis: Following Rep et al.'s interprocedural, finite, distributive, subset (IFDS) framework [8], we have also formulated a fully context-sensitive dataflow analysis as a grammar-guided reachability problem. Specifically, we use this dataflow analysis to track NULL value propagation. Under the IFDS framework, each dataflow fact corresponds to one vertex. The dataflow transfer function is represented as the relation mapping (edges) between vertices at different program points. Finally, an exploded supergraph is generated and the dataflow analysis is equivalent to the reachability computation on the graph. For more details, please refer to [8].

One slight difference between the IFDS framework and our formulation is that we achieve context sensitivity also by cloning intraprocedural graphs instead of using the summary-based approach in [8], which has been demonstrated [6] to fall short in answering many user queries.

2.3 Worklist Algorithm

Most of the state-of-the-art approaches [16], [9] implemented the CFL-reachability in a search fashion. Specifically, a worklist is maintained to store the reachable vertices. Each vertex is associated with the path information, i.e., the sequence of labels along the path leading to the vertex. The process is iteratively performed, each time a vertex from the worklist is popped off and the newly reachable vertices are added into the worklist by computing the path information. However, this search-based implementation is inherently unsuitable for parallelism, especially in our case the traversal is dynamically decided [17], [18]. In addition, as each path is traversed separately from start to end and none of the intermediate path information is cached for reuse, duplicated traversals are caused.

In addition, we reviewed a dynamic programming based CFL reachability algorithm [19] which all the intermediate edges are physically added. This algorithm maintains a graph $\mathcal{G}$ and a list $\mathcal{W}$ containing new edges. Each time, (1) one edge e is popped from $\mathcal{W}$; (2) then it generates new edges e' based on edge e and all its adjacent edges in $\mathcal{G}$ if the labels on the edges match a production rule in CFL like the example shown above in Figure 1; (3) it determines then if e' already exists in $\mathcal{G}$, if not, 4) add e' into both $\mathcal{W}$ and $\mathcal{G}$. It repeats the 4 steps until $\mathcal{W}$ becomes empty.

However, this dynamic programming based algorithm has significant drawbacks. Firstly, the traditional algorithm is sequential, which is difficult to be directly parallelized due to a large number of read and write operations on shared variables (list $\mathcal{W}$ and graph $\mathcal{G}$). The parallel implementation by means of locks suffers from unacceptable synchronization overheads. Similarly, in the distributed environment, such lock-based version (e.g., actor model-based implementation [20]) causes massive information to be transferred across networks, making it impractical. Secondly, the worklist algorithm needs to maintain a huge number of edges in memory. The limited memory size greatly hinders its applicability in practice.

In this work, we devise a data-parallel algorithm for CFL reachability, which is highly parallel and can leverage disks during computation.

3 A DISTRIBUTED SOLUTION

A naive parallel version of the dynamic programming algorithm [19] is to launch multiple threads to deal with multiple edges in

$\mathcal{W}$ simultaneously. Each thread handles one edge and performs the entire computation for it. In this way, we have to achieve synchronization by aid of locks, resulting in massive synchronization overheads. Even worse, in a distributed environment, as the communications are required between tasks, a great deal of message passing occurs, which leads to prohibitive communication cost. As a matter of fact, each task only plays with a small portion of the shared data. Fully locking of the shared variables is not necessary. For the sake of low synchronization overheads, we are intended to treat the shared data at a finer granularity.

Algorithm 1: A data-parallel worklist algorithm

```
   Data: The input graph $\mathcal{G}$ (a set of edges)
   Result: Updated graph $\mathcal{G}$ with new edges
1  $\mathcal{W} \leftarrow \mathcal{G}$
2  repeat
3      $\mathcal{P} \leftarrow \mathcal{G}$ join $\mathcal{W}$
4      $\mathcal{E} \leftarrow$ map ($\mathcal{P}$)
5      $\mathcal{W} \leftarrow$ distinct ($\mathcal{E}$ diff $\mathcal{G}$)
6      $\mathcal{G} \leftarrow \mathcal{W}$ union $\mathcal{G}$
7  until $\mathcal{W} == \emptyset$
8  return $\mathcal{G}$

9  Function map ($\mathcal{P}$)
10 $\mathcal{E} \leftarrow \emptyset$
11 foreach edge pair $x \xrightarrow{A} y \xrightarrow{B} z$ in $\mathcal{P}$ do
12     if a production rule C ::= A B exists then
13         $e \leftarrow x \xrightarrow{C} z$
14         $\mathcal{E} \leftarrow \mathcal{E} \cup \{e\}$
15 return $\mathcal{E}$
```

Different from the task-parallel algorithm, we regard the CFL reachability as a big data problem which is composed of a large dataset and the simple computation on that dataset. With proper modeling of this problem, we can then obtain an efficient solution with good performance and scalability through cluster computing. As such, we propose a data-parallel algorithm based on Map/Reduce-style operations, shown as Algorithm 1. The operations needed in our algorithm are commonly supported by many data-parallel distributed systems. Specifically, the process is executed in an iterative manner. Each iteration firstly performs a *join* on $\mathcal{G}$ and $\mathcal{W}$ to produce all the possible edge pairs, e.g., $x \xrightarrow{A} y \xrightarrow{B} z$ (Line 3). Next, a *map* operation is followed to perform the label matching on each edge pair (Lines 11-14) resulting in a variety of new edges $\mathcal{E}$ (Line 4). As the same edge could be generated in different ways multiple times, duplicated edges thus exist in $\mathcal{E}$. In order to avoid redundant computation thereafter, we need to filter out all the duplicates so as to obtain the newly added edges $\mathcal{W}$ (Line 5). These new edges are merged to the graph $\mathcal{G}$ via *union* for the next iteration (Line 6). The iterative process continues until no new edges are produced, i.e., $\mathcal{W}$ is empty (Line 7).

The above data-parallel algorithm is distinct from the task-parallel one in several aspects. Firstly, map/reduce operations intuitively partition the shared data into multiple small blocks. The data is processed in a divide-and-conquer manner, thus reducing synchronization overheads and achieving high parallelism. Secondly, we can choose the particular computation model according to the data locality to reduce the communication costs in the distributed environment. Finally, it is readily to utilize disks during computation by means of the mature distributed file systems.

4 Offline Batch Static Program Analysis System

We implemented the data-parallel algorithm in Section 3, and built an offline batch static program analysis system. This section first describes the computation model devised for implementing the data-parallel algorithm. Then a data structure with compression effect is presented, followed by some significant optimizations in system design.

4.1 Computation Model

We propose a *join-process-filter* computation model in our design illustrated as Figure 2. The whole computation is processed iteratively. At the beginning of each iteration, all the edges in both the graph G and worklist W are partitioned according to their source and destination vertices. Edges with the same source or destination vertex (i.e., the key) are put together into the same partition corresponding to the rectangle with gray background in Figure 2. For brevity, we only use two partitions for each dataset. Note that computations for different partitions are done in parallel.

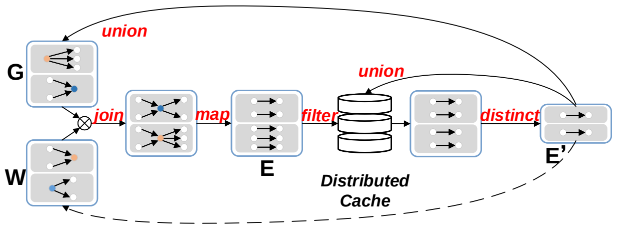


Fig. 2: Dataflow of BigSpa

Join: Once the partitioning is done, we next *join* all edge partitions resulting that two edges are gathered if the source vertex of one edge is identical to the destination of the other. As shown in Figure 2, the edges with the yellow source are connected together with edges of yellow destination forming multiple pairs of adjacent edges, e.g., $\langle x \rightarrow y, y \rightarrow z \rangle$.
Process: After joining, we obtain multiple partitions, each consisting of multiple pairs of adjacent edges. We next perform a *map* to process all the partitions in parallel. The computation required for each partition is simply to generate new edges by matching edge labels based on the grammar rules. To be more detailed, given an edge pair $x \xrightarrow{m} y \xrightarrow{VF} z$, a new edge $x \xrightarrow{OF} z$ will be produced according to the grammar rule (8) in Section 2.2.
Filter: When the map is completed, edges are generated in each partition shown as E. Duplicates exist in the sense that the same edge could be generated multiple times. For the purpose of correct termination and absence of redundant computation, we need to filter out all these duplicated edges. Our filtering consists of two phases: (1) *global filtering* which removes out from E all the edges already existing in G with a distributed cache system (For example, we can store G in a distributed database and query the elements in E from the database. Only those elements with empty query results will be retained.); (2) *local filtering* which gets rid of all the duplicated edges in E itself via a *distinct* operation.

After the entire *join-process-filter* procedure, a set of new edges E' is acquired. These new edges are repartitioned and merged into G. Simultaneously, we replace W with E' for the subsequent iteration. The computation terminates until no new edges are generated (i.e., E' is empty). Finally, G contains all the analysis result information which would be reported via a post-processing phase.

4.2 Data Structure

An edge in the program graph can be defined using a triplet (src, dst, label), which represents the source node, target node and label of the edge. However, this straightforward representation is not suitable for our computation model. First, simply using triplets to store edges is quite space-inefficient. A large amount of redundancy exists. Second, at each iteration, the $process$ part in Section 4.1 involves the join operation and label matching among edges, which requires expensive searches among a large number of triplets. Therefore, we design a compressed data structure with index optimization.

We use a key-value form to represent the edge set in a partition, with a vertex as the key and the set of edges connected to the vertex as the value. The basis for compression and optimization with indexes is that the symbols in the CFL are finite, which means there are only a limited number of types of edges. In practice, we can classify edges from three aspects:

1) Given a vertex, there are two directions for adjacent edges: incoming and outgoing;
2) According to the calculation state of the edge, there are two states: active (the edges to be processed) and silent (the edges that have already been processed);
3) Assuming that there are a total of L terminal and non-terminal symbols in the CFL, we have L kinds of labels.

Hence, all the edges associated with a certain vertex can be classified into $2 \times 2 \times L = 4L$ kinds.

Figure 3 shows an example of the structure, where the original edge set is compressed into two arrays (the indexes of arrays start from 1 in this figure). The array of neighbours contains the nodes adjacent to the intermediate node m. It is ensured that all the nodes of the same edge type are stored consecutively, and their starting position in the array is maintained respectively in the array of starting indexes. In this way, when searching for a particular type of edges connected to m, we are able to achieve an efficient random access by leveraging the index range information. For example, in Figure 3 if we are looking for the edges of type $[\mathcal{L}_k, outgoing, active]$, we simply look at the $[(k-1)*4+2+1]$th element in the array of starting indexes. Given $k=2$, since the 7th element in the index array is 11 and the following (8th) one is 12, we know that there is only one outgoing and active edge connected to m with label $\mathcal{L}_2$, i.e., $(m, v_{11}, \mathcal{L}_2)$.

Given that there are $4L$ types of edges, the length of the array of starting indexes is $4L$. The total storage size of the two arrays is $(4L+X)$. Therefore, when there are X edges and $X \gg 4L$, the amount of data stored is reduced from $3X$ to X. When the number of edges is large, the storage amount is reduced to one third (ignoring the platform implementation difference).

The index-based data structure not only reduces storage overheads, but also improves computational efficiency: when matching edges according to grammar rules, we can directly find the index range of the corresponding label with time complexity $O(1)$, instead of traversing the entire edge set.

4.3 Optimizations

Load balancing: Load balancing plays a crucial role in the performance of parallel computing. In our data-parallel algorithm, if the sizes of partitions vary significantly, it probably arrives at a status that the majority of threads finish the tasks quickly and become idle waiting for one thread, bringing about poor performance. In our computation model, for each vertex, all of its incoming and outgoing edges need to be visible to perform label matching and edge addition. For instance, when y is processed, both $x \xrightarrow{A} y$ and $y \xrightarrow{B} z$ need to be accessed to add the edge from x to z (shown as Line 11 to 14 in Algorithm 1). There perhaps exist some hot vertices which are associated with a huge number of adjacent edges. As a consequence, partitions would be unbalanced. As an example shown in Figure 4(a), we could get unbalanced partitions after joining – key 3 is hot, resulting in a large partition. Even though we could put both key 1 and 2 into one single partition, the tasks are still unbalanced (i.e., 3 edge pairs versus 6 edge pairs).

In order to address load unbalance, we adopt a finer-grained partitioning scheme. To be clear, we define the number of Matchable Edge-pairs (MEP) as Formula 11, where m is the intermediate vertex representing an edge-pair, and $E_m(direction, status, label)$ is the number of edges connected to m of a given type.

$$MEP(m) = \sum_{\mathcal{C}::=\mathcal{AB}\in Production} \begin{pmatrix} E_m(in, active, \mathcal{A}) \cdot E_m(out, all, \mathcal{B}) + \\ E_m(in, silent, \mathcal{A}) \cdot E_m(out, active, \mathcal{B}) \end{pmatrix} \tag{11}$$

The MEPs are calculated immediately after the $join$ process of each iteration. If the MEP of certain vertex is beyond some threshold, we split it into multiple small partitions. In practice, the threshold can be set to an empirical value or the average MEP of all the vertices. Since we conduct the edge generation and filtering separately in our *join-process-filter* model, splitting edge pairs of one vertex into multiple partitions does not affect the correctness of computation. The computation time cost on each node shown in Figure 14 empirically validated the effectiveness of our load balancing strategy.

Pre-shuffle: Pre-shuffle is intended to reduce the network traffic when a *join* is performed on two datasets of key-value pairs. The normal join shuffles all partitions from the two datasets over the network. The pre-shuffle avoids this network communication by partitioning two datasets with the same partitioner in advance and joining corresponding partitions directly in the same node without the network traffic. In BigSpa, the join of old edges and new edges triggers shuffle at each iteration. We maintain a global partitioner during the execution and assure that the old edges are always partitioned by the global partitioner. When new edges are generated, we partition it with the global partitioner and join it with old edges via pre-shuffle. The united dataset becoming the old edges in the next iteration inherits the global partitioner as its partitioner automatically. Figure 5 demonstrates the join process with pre-shuffle. In this way, the network communication is only involved during the repartitioning of the new edges at each iteration. BigSpa avoids re-shuffling old edges which become larger and larger, greatly reducing network traffic.

Computation closure: Computation closure is a design optimization to accelerate the convergence of edge addition for specific analysis workloads. We check the analysis grammar and find that certain edges labelled by terminal symbols are static during the iterations. We broadcast these edges to all the worker nodes so that these edges can be accessed locally. At each iteration, each node computes a local closure to generate as much more edges

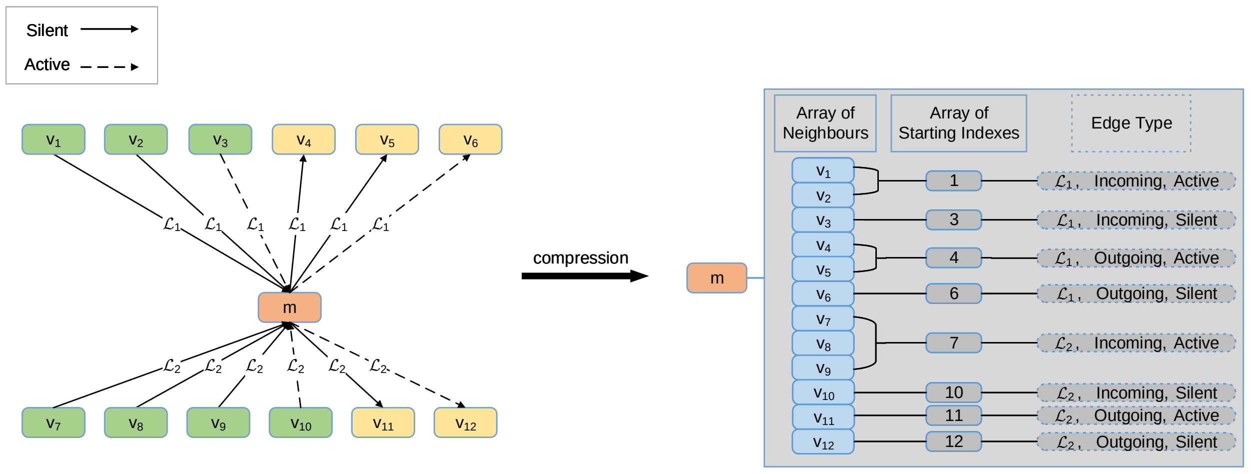

Silent
Active
v_1 v_2 v_3 v_4 v_5 v_6
m
v_7 v_8 v_9 v_{10} v_{11} v_{12}
compression
Array of Neighbours
Array of Starting Indexes
Edge Type
1 $\mathcal{L}_1$, Incoming, Active
3 $\mathcal{L}_1$, Incoming, Silent
4 $\mathcal{L}_1$, Outgoing, Active
6 $\mathcal{L}_1$, Outgoing, Silent
7 $\mathcal{L}_2$, Incoming, Active
10 $\mathcal{L}_2$, Incoming, Silent
11 $\mathcal{L}_2$, Outgoing, Active
12 $\mathcal{L}_2$, Outgoing, Silent

Fig. 3: Key-value Pair Data Structure with Index Compression

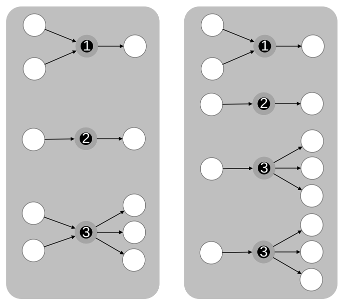


Fig. 4: Load balancing by splitting

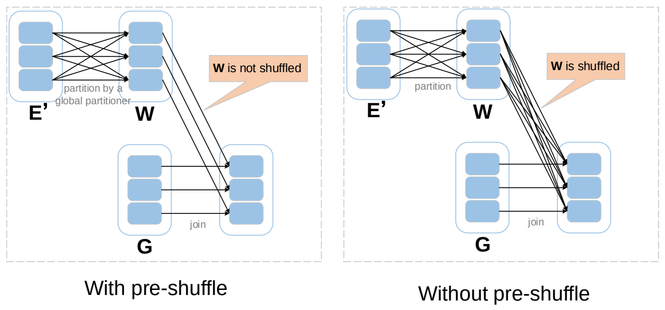


Fig. 5: Join Process with Pre-shuffle

as possible with the help of the broadcasted static edges. Take the dataflow analysis as an example, there exists one production rule $N ::= N \enspace e$ in the grammar. The edges labelled by terminal e are static. In other words, no new e edges will be added during computation. Given a path in the input graph $x \xrightarrow{N} y \xrightarrow{e} z \xrightarrow{e} w$, we should get two new edges $x \xrightarrow{N} z$ and $x \xrightarrow{N} w$ generated in the end. In our computation model, the new edge $x \xrightarrow{N} z$ is first generated by matching labels on the edge pair $x \xrightarrow{N} y \xrightarrow{e} z$ based on the grammar rule. However, the generation of new edge $x \xrightarrow{N} w$ will be postponed until $x \xrightarrow{N} z$ and $z \xrightarrow{e} w$ are joined together in the future iteration. In fact, the edges labelled by e can be broadcasted to all worker nodes at the very beginning. To this end, we are able to generate as many new edges as possible locally in fewer iterations. With the aid of this optimization, we reduced the number of iterations from a few thousands to a hundred in certain analyses.

5 Online Incremental Static Program Analysis System

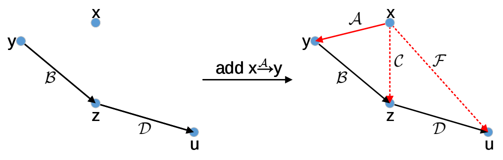
In the previous section, we introduced how to process large scale static program analysis over the entire code dataset in the batch processing way. In fact, in the real world, software codebases can be updated day by day, especially for those popular open source software with large scale developer communities. In this scenario, when a program to be analyzed is changed or updated, re-analyzing the entire graph can be time-consuming and cumbersome.

Instead, the system can perform local analysis on the updated parts to achieve high efficiency in time and memory usage. To handle this scenario, besides the above offline batch static program analysis system, we further design an online incremental static program analysis system. In this section, we first analyze the incremental update process, and then introduce the computation model and data structure specifically designed for the online incremental system, followed by an adaptive computation mode selection mechanism.

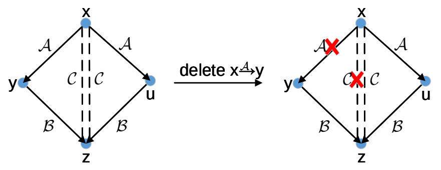
5.1 Process of Incremental Update

Similarly, edges in the program analysis graph represent the reachability relationships between vertices, and there are two types of them: directly reachable (those formed by directly adding edges to the graph) and indirectly reachable (those generated by label matching based on the CFL syntax). When adding or deleting edges, the indirect reachability relationships generated by these edges should be changed accordingly. Therefore, an update on the graph requires iterative processing until the topology of the graph reaches a stable state.

In order to improve the performance, we use the mini-batch update strategy to perform incremental static analysis, that is, multiple edges are added to the graph at the same time, and then the updated transitive closure is computed. During the mini-batch update process, although the order of adding and deleting



(a) Iteratively adding an edge



(b) Iteratively deleting an edge

Fig. 6: Iterative Process of Adding and Deleting an Edge

edges may change, the graph topology of the final update graph is determined. Obviously, the same topology will lead to the same reachability relationship. This ensures the correctness of our mini-batch update method.

5.2 Method for Online Incremental Analysis

5.2.1 Triangle Count

As shown in Figure 6, the difficulty of deleting edges is that an indirect edge could be generated on multiple paths. Without extra information, we cannot decide whether to delete that indirect edge or not. A straightforward approach is to save all the generating paths so that we can distinguish the same edge on different paths, but this will greatly increase the storage and calculation overheads (as discussed in *Filter* part in Section 4.1). To deal with the edge deletion correctly and efficiently, we propose a method based on the triangle count (short as tri_count). In this paper, we define the triangle count of an edge e as the number of triangles with e as one of their sides. Essentially, triangle count ($tri\_count(x \xrightarrow{\mathcal{C}} y)$) is the number of two-hop paths from x to y in the reachability graph. The calculation of the triangle count is shown in Formula 12.

$$tri\_count(v_1 \xrightarrow{\mathcal{C}} v_3) \\ = |\{v_2 | \exists(v_1, v_2, \mathcal{A}) \in TC \,\&\&\, \exists(v_2, v_3, \mathcal{B}) \in TC \,\&\&\, \mathcal{C} ::= \mathcal{AB}\}| \tag{12}$$

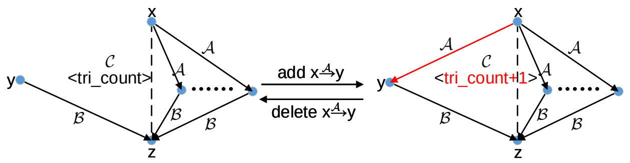

x
y
z
$\mathcal{C}$
<tri_count>
$\mathcal{A}$
$\mathcal{B}$
add x$\xrightarrow{\mathcal{A}}$y
delete x$\xrightarrow{\mathcal{A}}$y
<tri_count+1>

Fig. 7: Addition and Deletion Process Based on Triangle Count

Figure 7 describes the edge addition and deletion process based on the triangle count method, and the operations are quite similar. For example, if an edge $x \xrightarrow{\mathcal{A}} y$ is added, the system matches edge pairs based on labels and trigger the addition of an indirect edge $x \xrightarrow{\mathcal{C}} z$. When the edge $x \xrightarrow{\mathcal{A}} y$ is deleted, the same label matching process is performed and we can find the same indirect edge, which was generated by this edge and should be affected by the deletion. When a new indirect edge is newly generated, its tri_count is initialized to 1. During the iterative calculation, each time an addition/deletion of an indirect edge is triggered, the tri_count of this edge is increased/decreased by one. An edge is removed from the graph only when its tri_count is reduced to zero. In this way, when updating the transitive closure, the system can decide whether to add/delete an indirect edge or just change the tri_count of an edge. A theoretical proof for the correctness of using triangle counts to determine reachability can be found in Appendix.

5.2.2 Online Incremental Computation Algorithm

Algorithm 2 shows the procedure of online incremental analysis. Compared with algorithm 1, the main change is that a new variable $\mathcal{W}_{count}$ is defined, which stores the newly generated edges and the number of times they are added or deleted. It is used to update the triangle counts and generate the edge set $\mathcal{W}$ to be calculated. During each iteration, $\mathcal{W}_{count}$ is updated to ensure that it only covers the additions and deletions generated within the current round of computation, so that no triangles are missed or double-counted.

Algorithm 2: Online incremental computation algorithm

```
Data: A batch of updates on graph $\mathcal{G}$, Production set $\mathcal{P}$
Result: TC of $\mathcal{G}$
1 $\mathcal{W}_{count} \leftarrow count\ \ batch$
2 $\mathcal{W} \leftarrow \mathcal{W}_{count}$ diff $TC$
3 repeat
4     $\mathcal{P} \leftarrow TC$ join $\mathcal{W}$
5     $Update\ \ TC\ \ with\ \ \mathcal{W}_{count}$
6     $\mathcal{W}_{count} \leftarrow$ map $(\mathcal{P})$
7     $\mathcal{W} \leftarrow \mathcal{W}_{count}$ diff $TC$
8 until $\mathcal{W} == \emptyset$
9 $Update\ \ TC\ \ with\ \ \mathcal{W}_{count}$
```

The computation model is thus modified as Figure 8. In the online processing system, in order to get a lower response time for queries and updates, the distributed databases are utilized not only for filtering, but also for storing and updating the graph. The *prediction* part in the figure will be further discussed in Section 5.4.

5.3 Data Structure

To record the triangle counts of each edge and reach the requirement of real-time response, we then improve the data structure introduced in Section 4.2. As shown in Figure 9, according to their labels (assuming there are L types of labels) and directions (inside and out), the adjacent edges of each intermediate node are divided into $2L$ subsets. Note that the edge calculation state is not considered here, as all edges stored in the online system are silent edges (i.e., the reachable relationships that have already been calculated).

In general, iterative calculations caused by incremental updates are local, which means there is no need for the whole graph or all adjacent edges of a node to be involved in the calculation. With this fine-grained data structure, the system can access data on demand when updating the graph. Besides, there are two more advantages to storing the adjacent edges separately: firstly, it can avoid a single record in the database being too large; secondly, the system can directly locate the edges needed for calculation, reducing the cost for intermediate results.

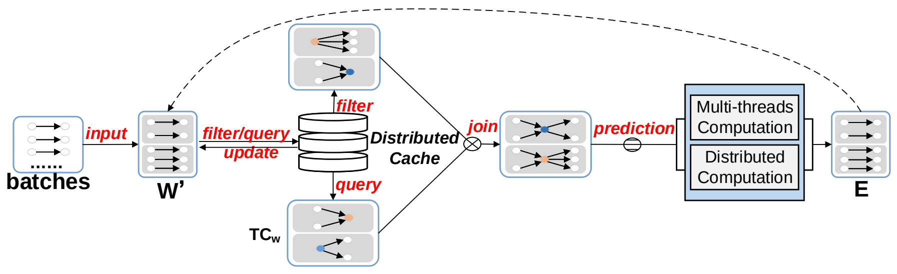


Fig. 8: Dataflow of Online Incremental Analysis System

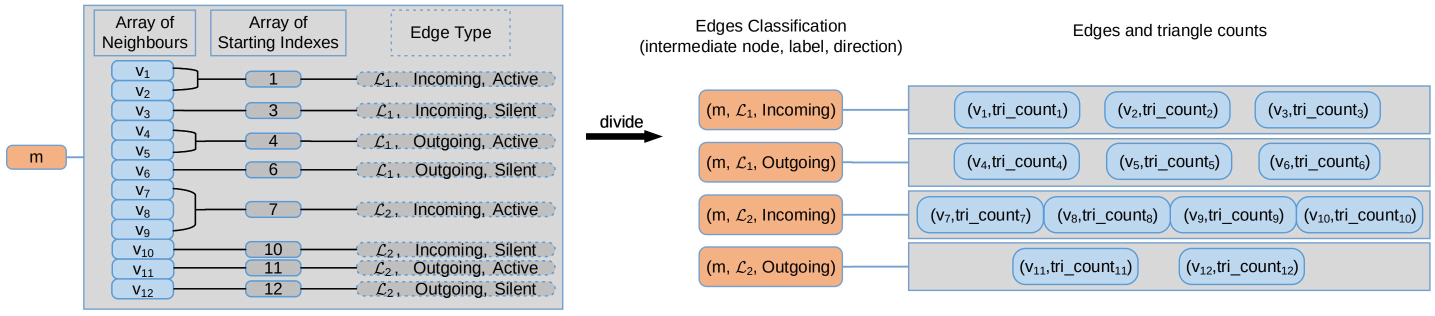

m
Array of Neighbours
Array of Starting Indexes
Edge Type
v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12
1 3 4 6 7 10 11 12
L1, Incoming, Active
L1, Incoming, Silent
L1, Outgoing, Active
L1, Outgoing, Silent
L2, Incoming, Active
L2, Incoming, Silent
L2, Outgoing, Active
L2, Outgoing, Silent
divide
Edges Classification (intermediate node, label, direction)
Edges and triangle counts
(m, L1, Incoming)
(m, L1, Outgoing)
(m, L2, Incoming)
(m, L2, Outgoing)
(v1,tri_count1) (v2,tri_count2) (v3,tri_count3)
(v4,tri_count4) (v5,tri_count5) (v6,tri_count6)
(v7,tri_count7) (v8,tri_count8) (v9,tri_count9) (v10,tri_count10)
(v11,tri_count11) (v12,tri_count12)

Fig. 9: Edge Set Storage Data Structure of Online Incremental Analysis System

5.4 Adaptive Computation Mode Switching Mechanism

The online incremental analysis system is designed to deal with small batches of update data, which, in most cases, trigger a small amount of computation. However, in some scenarios, small batch updates can also trigger large-scale calculations (as is the case where a large number of indirect edges are generated, mentioned in Section 4.3). Distributed platforms are suitable for the latter case, but they can be greatly slowed down by transmission between computing nodes in the former case. Considering that, we design an adaptive mode switching mechanism. Based on the prediction of computation scale, the system can automatically switch between single-machine multi-threaded computing mode and distributed computing mode.

In online incremental processing, after the join operations, a calculation scale prediction process is performed. Specifically, we calculate the sum of MEP (defined in Section 4.3) of all the intermediate nodes, which is the number of new edges to be generated and can indicate the scale of calculation. Hence, if the sum of MEP is below a threshold, the map process will be performed in the single-machine multi-threaded mode, otherwise the data to be processed is divided into different partitions and the distributed computing mode is activated. During each iteration, the system performs the prediction and switches between the two modes flexibly. In this way, we can both avoid computational failure due to insufficient single-node resources and reduce the resource waste and timeout due to excessive overheads in distributed platforms.

6 System Design and Usage

6.1 System Architecture

The overall system architecture of BigSpa is shown in Figure 10, which consists of four layers:

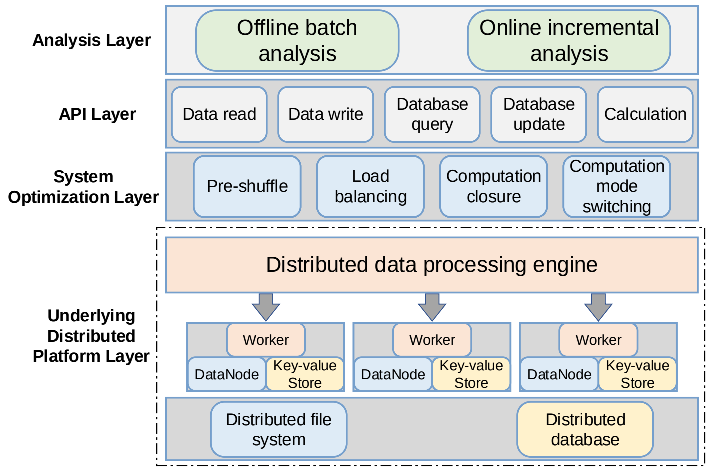


Fig. 10: System Structure of BigSpa

- Analysis Layer: The top layer of the program static analysis system includes the offline batch and online incremental analysis system described in Section 4 and Section 5.
- API Layer: There are five types of operators to complete specific reading and calculation tasks for application usage.
- System Optimization Layer: This layer implements the various optimization measures, such as pre-shuffle, load balancing, computation closure and computation mode switching mechanism.
- Underlying Distributed Platform Layer: This layer includes a distributed data processing engine, a distributed file system, and a distributed database. In implementation, we choose

Spark as the data processing platform, Hadoop HDFS as the distributed file system, and Redis as the distributed database.

6.2 Implementations

We implemented BigSpa on top of Apache Spark and Redis.

Apache Spark: Apache Spark is a fast and general-purpose distributed computing system. Spark uses the RDD (Resilient Distributed Dataset) as its data abstraction, which is operated in parallel. RDDs can be persisted in memory for reuse during iterations to achieve high performance, or be processed through disks for scalability if the data can not fit into the memory. A RDD consists of multiple partitions, each of which is processed by a single thread. RDDs support a series of operations, such as map, filter, join, union, distinct and so on. The operations on RDDs are all conducted in parallel – each thread performs the operation on a partition of the RDD.

Within our implementation, all the edge data are organized in the RDD format corresponding to each rectangle in Figure 2. The small rectangles with grey background denote partitions within a RDD. All the operations needed in our computation model are directly implemented by operations of RDDs.

Redis: Redis is an advanced key-value store that can be applied as a database or a cache. In order to achieve the high performance, Redis stores the dataset in memory and allows the users to persist the data on disk periodically. In BigSpa's implementation, we leverage Redis to filter out massive duplicated edges by organizing the individual Redis servers in the cluster into a unified distributed key-value database.

Note that the implementation of our data-parallel algorithm and computation model is not restricted by Apache Spark and Redis. We can actually adopt any other distributed data-parallel infrastructures, like Apache Hadoop, Apache Flink etc. and distributed cache systems, e.g., Apache HBase, Memcached for implementation.

6.3 BigSpa Usage

Figure 11 demonstrates how to use BigSpa to carry out various analyses. Analogous to the declarative program analysis [15], [21], [22], we separated the computation back-end from the client analysis implementations. As stated earlier, BigSpa as a general analysis computation engine supports multiple analyses[10]. In order for users to implement a particular sophisticated interprocedural analysis, a front-end is required. As shown in Figure 11, we implemented two front-ends, one for dataflow and the other for pointer analysis (both field-insensitive and field-sensitive) in our experiments. Users can develop their own front-ends for specific client analyses and then leave the computation to BigSpa. In the following, we discuss the two tasks of which each front-end is comprised, namely generating the program graph and specifying the analysis grammar.

Generating Graphs: In order for BigSpa to perform an interprocedural analysis, users are first required to generate a specialized program graph tailored for the analysis.

For example, the graph generator of our pointer/alias analysis first emits the intra-graph similar to the program expression graph presented in [16]. Differently, in our graph generation, we achieve the context sensitivity via function inlining. We conduct a bottom-up traversal of the call graph to inline functions. The intra-graph of a function is cloned at each call site invoking that function.

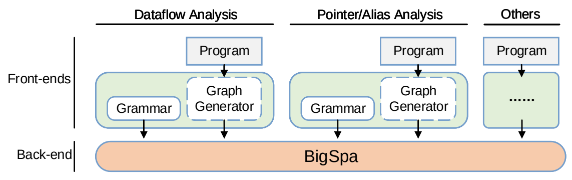


Fig. 11: BigSpa Usage

Eventually, the inter-graph is obtained by connecting the intra-graphs through edges between formal and actual parameters.

Specifying Grammar: Given the graph generated, the front-end needs to specify a particular grammar guiding the edge addition for the analysis. For example, pointer/alias analysis could take advantage of the grammar similar to the rules 8 - 10 in Section 2.2.

For the purpose of performance, our model is devised to access at most two adjacent edges for label matching. Such simple computation logic makes our design easier and leads to a dynamic programming style algorithm which imposes lower complexity. To this end, each production rule in the grammar needs to be normalized so that it has no more than two terms on its right-hand-side (RHS). The transformation can be done by introducing new non-terminal symbols in linear time, and causes a linear increase in the size of the grammar [19]. The normalized grammars for field-insensitive pointer/alias analysis and dataflow analysis can be found in Appendix.

7 Evaluation

This section presents an empirical evaluation of BigSpa. We focus on three research questions:

- Q1: What is the performance of BigSpa?
- Q2: How does BigSpa compare to the state-of-the-art single-machine analysis system?
- Q3: How does BigSpa compare to other state-of-the-art distributed systems for analysis workloads?

Hardware and Software Environment: Unless otherwise specified, experiments presented in this paper were conducted in a cluster with 17 nodes (1 master node + 16 worker nodes). Each node was equipped with two Intel Xeon E5620 CPUs (2.4GHz/12MB Cache, 12 physical cores and 24 logical cores in total), 16 GB available memory and 3 TB HDD RAID0 disk storage. All nodes were connected via 1Gbps Ethernet. Our system was implemented with Apache Spark 2.2.0 and used Apache Hadoop 2.7.2 to provide HDFS and Yarn. Redis 3.2.0 was employed as the distributed database. We compiled and ran all the programs with JDK 1.8.0 and Scala 2.11.8 on the RHEL 7 64-bit OS with Linux kernel v3.10.0.

Subject Programs & Analyses: We selected five large-scale real-world codebases namely Linux kernel, PostgreSQL database, Apache HTTP server, Apache Hadoop HDFS, and Apache Hadoop MapReduce as our analysis subjects. Table 1 shows the detailed characteristics of the subjects including the version information, the numbers of lines of code, the description and implementation language.

Three interprocedural analyses were mainly performed, namely context-sensitive field-insensitive Andersen's inclusion-based pointer/alias analysis, context-sensitive field-sensitive pointer/alias analysis, and dataflow analysis for null-pointer propagation, which are denoted in abbreviation as pointer/alias analysis, field-sensitive pointer/alias analysis and dataflow analysis, respectively. We focused on these three client analyses since they are the well-known representatives of a wide range of analyses that can be formulated as a grammar-guided graph reachability problem [10]. As the null-pointer does not make much sense for Java programs, we ignored the dataflow analysis for the subjects HDFS and Hadoop-MapReduce. Similarly, we mainly evaluated the field-sensitive analysis on Java subjects. We achieved the fully context sensitivity by cloning function bodies for every single context [23].

Table 2 lists the number of nodes with different degrees in the input graphs. All of these graphs have over millions of edges and some of them are heavily skewed.

TABLE 1: Characteristics of Subject Programs

Subject	Version	#LoC	Description	Language
Linux	4.4.0-rc5	16M	Operating system	C/C++
PostgreSQL	8.3.9	700K	Database	C/C++
httpd	2.2.18	300K	Web server	C/C++
HDFS	2.0.3	546K	distributed file system	Java
Hadoop-MR	2.7.5	568K	data processing platform	Java

TABLE 2: Degree Distribution of Input Graphs

Degree range	0-1000	1000-5000	5000-1E4	1E4-5E4	>1E5	Total
Pointer/Alias Analysis						
Linux	52,877,027	409	0	0	0	52,877,436
PSQL	5,203,335	84	0	0	0	5,203,419
httpd	1,721,413	5	0	0	0	1,721,418
HDFS	5,340,286	111	35	26	4	5,340,468
Hadoop	21,862,215	1,381	67	14	22	21,863,699
Dataflow Analysis						
Linux	65,210,594	90	0	0	0	65,210,684
PSQL	5,743,046	1,183	0	0	0	5,744,229
httpd	29,819,944	463	0	0	0	29,820,407
Field-sensitive Pointer/Alias Analysis						
HDFS	7,173,123	3,993	119	25	0	7,177,260
Hadoop	6,151,761	1,688	50	10	5	6,153,514

Since the analyses in our experiments have considered the highest level of context sensitivity, and the precision together with the effectiveness of such analyses has been already validated in prior work[7], [24], we mainly focus on the efficiency and scalability of BigSpa in our evaluations.

7.1 Evaluation of Offline Batch System

We first evaluate the performance of the offline batch system.

7.1.1 Performance Analysis

Performance and Scalability: Table 3 reports the performance data of BigSpa. The columns #V(M), #EB(M), #EA(M), #Ite., PT(mins), CT(mins), and TT(mins) correspond to the number of vertices, number of edges before computation, number of edges after computation, the number of iterations, the preprocessing time, the computation time and the total time, respectively. All the numbers of vertices and edges reported are in millions and the time costs are in minutes. The preprocessing time is mainly taken by loading graphs and grammars into the distributed file system (HDFS) and the cache system (Redis), while the computation time shows the time cost for edge addition.

BigSpa managed to finish most of the analyses within 20 minutes in our experiments. Even for the biggest graph with more than a billion edges, BigSpa only took around half an hour

TABLE 3: The Peformance of BigSPA in Batch Processing

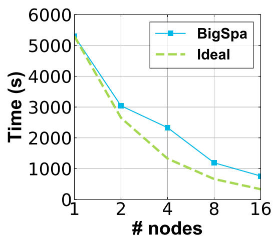
Subj.	#V(M)	#EB(M)	#EA(M)	#Ite.	PT(mins)	CT(mins)	TT(mins)
Pointer/Alias Analysis							
Linux	52.88	249.50	1.13B	73	2.51	27.3	29.81
PSQL	5.20	14.97	862.18	70	0.75	11.6	12.35
httpd	1.72	8.19	904.34	65	0.63	12.05	12.68
HDFS	5.34	19.10	1.80B	7	1.15	25.46	26.61
Hadoop	21.88	77.17	99.15	9	1.65	3.25	4.9
Dataflow Analysis							
Linux	63.02	69.41	137.48	112	6.36	8.95	15.31
PSQL	29.04	34.80	56.08	59	5.36	4.62	9.98
httpd	5.33	10.04	19.30	23	1.4	2.02	3.42
Field-sensitive Pointer/Alias Analysis							
HDFS	7.17	25.19	486.83	5	3.43	81.7	85.13
Hadoop	6.15	17.76	42.86	3	4.47	23.17	27.64

to complete the computation. In the field-sensitive pointer/alias analysis, as each of the grammars contains thousands of symbols, the intermediate results take up extremely large amount of memory even with our compressed data structure. We increased the available memory of each node to 30 GB and completed the analysis of HDFS and Hadoop with 25 nodes and 32 nodes respectively. It took BigSpa tens of minutes to finish the analyses. By contrast, the cutting-edge single machine systems like Graspan [7] were unable to process neither of the above two workloads, because they greatly exceed the processing capability of one single machine.

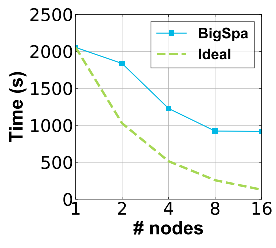
Table 4 shows the proportion of time spent on updating database, shuffling data between machines and Java's garbage collection on the representative subjects. For pointer/alias analysis, the computation time dominated the overall time cost. The communication was mainly spent on updating Redis, as the new edges generated in each iteration were written into the database. As for dataflow analysis, the preprocessing time became comparable with the computation time. The preprocessing time was mainly spent on broadcasting edges which is required by the closure optimization strategy discussed in 4.3. This also greatly reduced the number of edges that need to be written into Redis. GC took much more time when analyzing large or heavily-skewed graphs, since the computation process generates more intermediate objects.

TABLE 4: Proportion of Time Spent on Communications and Garbage Collection

Subj.	Redis-ratio (%)	Shuffle-ratio (%)	GC-ratio (%)
Pointer/Alias Analysis			
Linux	18.63	4.15	9.41
PSQL	35.12	2.71	5.07
httpd	35.17	1.99	7.62
HDFS	31.43	4.80	37.94
Hadoop	2.85	7.15	12.93
Dataflow Analysis			
Linux	4.37	5.80	46.80
PSQL	4.05	8.27	32.48
httpd	4.66	8.49	8.83



(a) Pointer/Alias Analysis on httpd



(b) Dataflow Analysis on Linux

Fig. 12: Scalability Analysis

Figure 12 plots the analysis time of BigSpa running on a cluster with varying number of nodes. We present here the results of two analyses which can be finished in reasonable time even on a single node. The detailed data for all subject analyses can be checked shortly in Table 6. We can read from the trend that BigSpa scales well (near linear) with the number of nodes available.

CPU and Memory Usage: Figure 13 illustrates the CPU and memory usage of a node when we ran BigSpa in a cluster for PostgreSQL. The X axis represents the iterations. The left and right Y axes denote the average CPU utilization and memory consumption of the node, respectively. The memory occupied by each machine was composed of three parts: (1) Spark RDD memory, which stores the edge set and the intermediate results; (2) Redis memory, which stores the transitive closure for the $filter$ process; (3) other memory reserved by Hadoop and Yarn, etc.

For pointer/alias analysis, the CPU utilization is relatively low and the memory consumption increases as BigSpa iterates. This is consistent with Figure 17 in the sense that the *map* stage contributes little to the overall cost. It further confirms that the pointer/alias analysis is network-communication intensive in BigSpa. For dataflow analysis, the memory consumption is steady since the number of new edges generated in the dataflow analysis is relatively small. Besides, in dataflow analysis Redis occupies only a small amount of memory. This is because only a few new edges are generated in the dataflow analysis and the Redis overhead allocated to each machine is minimal. The CPU utilization varies a lot, which has a strong correlation with the number of new edges added at each iteration. We only show the data for PostgreSQL here. Other subjects have the similar trends.

Load Balancing: We measured the execution time of *map* phase (i.e., computation) at each node in the cluster. Figure 14 presents the data for pointer/alias analysis on Linux kernel. The results for other analyses show similar trend. X axis denotes the computing node ID in the cluster, while Y axis shows the execution time in seconds. The dashed line corresponds to the average time (56.21 seconds) across the whole cluster. We computed the standard deviation which is as low as 1.4% (0.81 / 56.21). Figure 15 provides box-plots showing the deviation of the mean CPU usage of all cores of a node. It can be seen that during each iteration, the 24 CPU cores on the same node have no obvious load difference and the maximum CPU usage difference is about 10%. These results empirically validate that our load balancing strategy works well on both node level and thread level.

Optimizations: In this part, we evaluate the efficiency of the optimizations described in 4.3. Different optimizations work for different scenarios: the pre-shuffle and load balancing optimizations are tailored for pointer/alias analysis, while the closure optimization is for dataflow analysis.

Figure 16 illustrates the effects of pre-shuffle and load balancing on the total analysis time. The optimizations can significantly lower the end-to-end time cost in general because the pre-shuffle strategy greatly reduces data transfer overhead, especially when a large number of new edges are generated. Apart from this, the node splitting operation can prevent too many computing tasks from being assigned to a single node. The improvement brought by load balancing is particularly obvious on some extremely skewed graphs like HDFS. Without this, it leads to much longer processing time or even failures. The only exception is pointer/alias analysis on Hadoop, where the optimizations introduce an additional time cost (about 30 seconds). Since the scale of graphs for this benchmark is small, the optimizations can only make a relatively small improvement but still introduce a little extra cost.

TABLE 5: Effect of Computation Closure Optimization

Analysis	Subject	#Ite.Before	#Ite.After	Runtime Before(mins)	Runtime After(mins)
dataflow	Linux	6636	112	933.31	15.31
	PSQL	721	59	58.93	9.98
	httpd	181	23	8.55	3.42

Table 5 lists the effect of closure optimization on dataflow analysis. Although the dataflow analysis generates a relatively small number of edges compared with pointer/alias analysis, it requires much more iterations and thus takes much longer computation time. The optimization of computation closure can greatly reduce the number of iterations needed, especially on the Linux dataset, where the number of iterations is reduced from 6636 to 112 (a reduction of 98.31%), saving hundreds of hours of calculation time.

Time Cost Distribution: We monitored the time cost distribution and number of edges added at each iteration of BigSpa for each analysis shown as Figure 17. Within each iteration, the time cost comes from five stages, namely join, map, filter, distinct, and union corresponding to each stage in our computation model (Figure 2).

For pointer/alias analysis (Figure 17(c)), the time cost at each iteration ascends to a peak and then descends. After the peak, there is a long tail to converge. In addition, as shown in the plots, there exists a high correlation between the time cost and the number of edges added (indicated by the solid line) at each iteration. The trend of the *join* time verifies the efficiency of the pre-shuffle optimization discussed in 4.3. As BigSpa iterates, the total number of edges increases and the time spent on join should also increase. The pre-shuffle optimization makes BigSpa only shuffle newly added edges at each iteration, thus stabilizing the time cost at the join stage. Note that although the join time cost does not increase as the computation proceeds, it still dominates the total time cost in pointer/alias analysis.

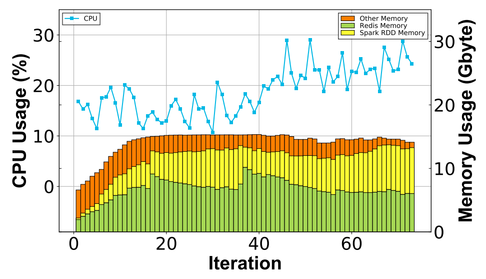
The dataflow analysis (Figure 17(d)) has the similar trend to the pointer/alias analysis but with the peaks appearing at the beginning. As the dataflow analysis exploits the computation closure optimization, a large number of new edges are quickly added in the first few iterations.

7.1.2 Comparison with Graspan[7]

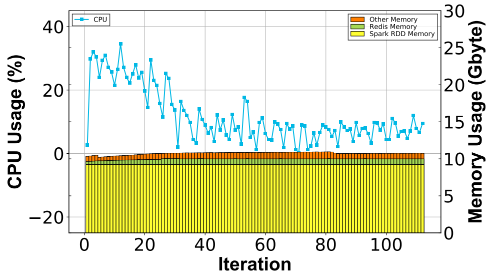
We compared BigSpa against a single-machine disk-based inter-procedural analysis system, called Graspan [7]. BigSpa was experimented with 1, 2, 4, 8, 16 nodes successively. As a comparison, we ran C++ version Graspan on one single node equipped with the same CPU as described before. In our experiments, Graspan was configured to use 24 threads, 4 initial partitions and 16 GB memory. Since it is an out-of-core system, we conducted Graspan experiments on both HDD RAID0 disks and SSD RAID0 disks. Table 6 shows the detailed running time in minutes and the COST metric[*].

Because Graspan's IO time accounts for only a small proportion of the total cost, and most of its disk reads and writes are sequential, using SSD can only bring little acceleration. With 16 nodes, BigSpa runs one or more orders of magnitude faster than Graspan in most cases. However, for certain analyses especially

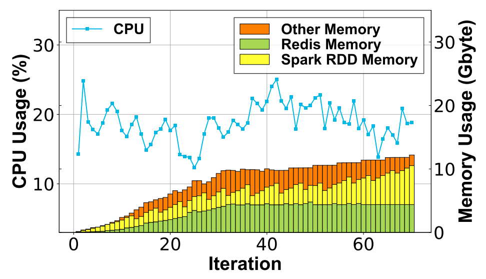
*. The COST proposed by [25] is used to compare a platform for a given problem with a competent single-threaded implementation. Here we use it to indicate the number of nodes required by BigSpa to outperform Graspan.



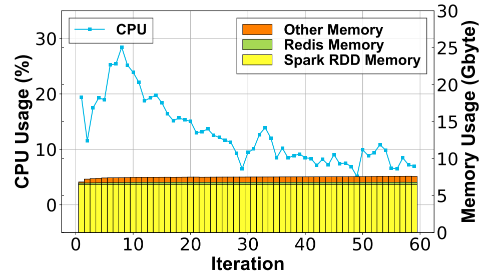
(a) Pointer/Alias Analysis For Linux



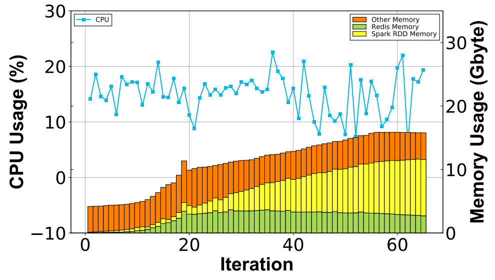
(b) Dataflow Analysis For PSQL



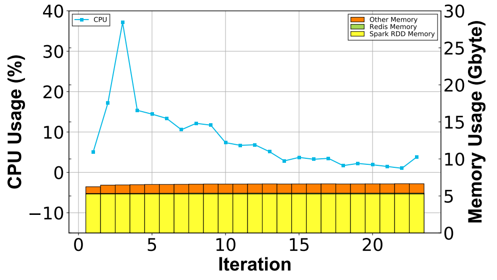
(c) Pointer/Alias Analysis For PSQL



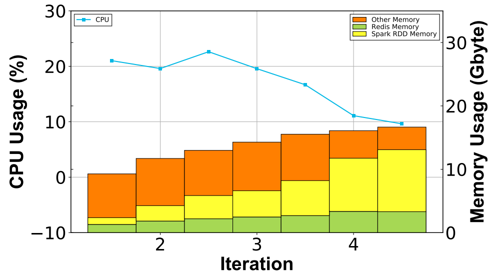
(d) Dataflow Analysis For PSQL



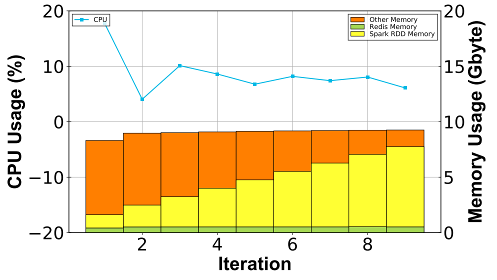
(e) Pointer/Alias Analysis For PSQL



(f) Dataflow Analysis For PSQL



(g) Pointer/Alias Analysis For HDFS



(h) Pointer/Alias Analysis For Hadoop

Fig. 13: CPU Utilization and Memory Consumption for PostgreSQL

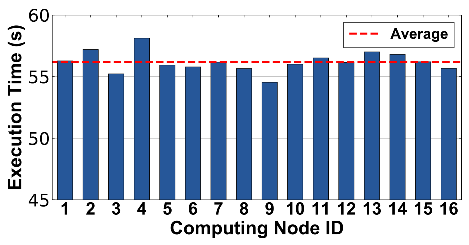


Fig. 14: Execution Time for Each Computer Node in the Cluster

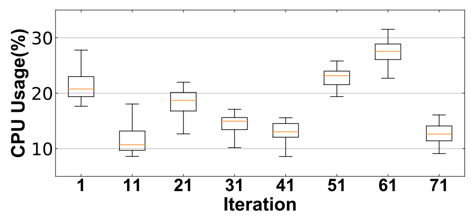


Fig. 15: Deviation of the Mean CPU Usage of All Cores of a Node

with large graphs, the performance of BigSpa running on a few (1 or 2) nodes is disappointing. The underlying reason is that BigSpa based on Apache Spark utilizes memory as higher priority for efficient computation. Unfortunately, once the memory consumption exceeds certain limit, heavy GC (Garbage Collection), frequent

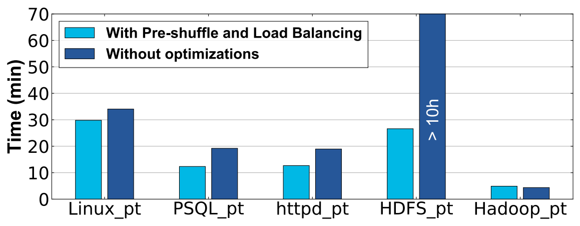


Fig. 16: Effect of Pre-shuffle and Load Balancing

TABLE 6: Performance Comparison with Graspan[7] In Minutes

	BigSpa					Graspan		COST*
Subject	1	2	4	8	16	SSD	HDD	-
Pointer/Alias Analysis								
Linux	>10hrs	253	120.97	47.14	29.81	142.55	150.02	4
PSQL	92.36	74.98	38.38	21.6	12.35	532.68	535.70	1
httpd	88.35	50.69	38.86	19.83	12.68	542.28	545.83	1
HDFS	>10hrs	>10hrs	102.75	80.51	26.61	>10hrs	>10hrs	4
Hadoop	25.6	14.06	9.59	6.4	4.9	69.55	70.52	1
Dataflow Analysis								
Linux	34.22	30.6	20.45	15.36	15.31	482.38	510	1
PSQL	17.63	14.27	13.9	8.89	9.98	118.87	123.85	1
httpd	8.62	6.78	6.8	5.82	3.42	7.57	7.6	2

disk I/O and data serialization/de-serialization on massive data have to be involved resulting in unacceptable performance. The experiment running on a single node reported that, the maximum I/O could reach 200 MB per second. The GC time even occupies more than 70% of the overall execution time at late iterations. That explains why a super-linear speedup appeared when analyzing large-scale pointer analysis graphs. As more computer nodes become available, memory pressure is relieved. GC time hence drops dramatically, achieving superlinear speedup.

Note that BigSpa performs significantly well on dataflow analysis compared against Graspan. BigSpa running on one node still outperforms Graspan a lot. The performance advantage mainly gives the credit to the following two design optimizations. First, BigSpa adopts computation closure optimization to reduce the number of iterations needed, which saves a great amount of time for convergence. Second, duplication removal and edge union are implemented in a more efficient way in BigSpa than Graspan. Graspan applies the merge operation to filter out duplicated edges and to unite edges. The cost of merge operation depends on the numbers of both new and old edges. Even though just a few edges are newly generated, Graspan still needs to reconsider a great number of old edges. BigSpa exploits the Redis database to conduct the filter operation and adopts pre-shuffle technique to join the new edges with the old ones efficiently. The costs of both operations are just related to the number of newly added edges.

In the experiments for field-sensitive pointer/alias analysis, BigSpa needed at most 32 nodes to finish proncessing the HDFS and Hadoop data sets in dozens of minutes (the detailed performance data can be found in Table3). By contrast, Graspan failed to run on both HDFS and Hadoop data sets because the scale of both data sets greatly exceeded the processing capability of one single machine. Although the process of parallelization makes BigSpa introduce additional overhead for a single machine, it provides a more scalable solution for inter-procedural analysis. Users can utilize BigSpa to perform large-scale sophisticated analysis tasks simply by scaling the system. Moreover, with additional resources and the optimizations enabled, BigSpa managed to process the heavily skewed graphs and generated millions of edges within a few iterations, demonstrating the advantages of distributed computing.

7.1.3 Comparisons with Other Distributed Systems

We also conducted the comparisons with other distributed systems on static analysis workloads, namely BigDatalog[21], and distributed call graph analysis[20].

Comparison with BigDatalog: BigDatalog[21] is a distributed datalog-based[15] computation system which supports efficient static analysis. We compared the performance of BigDatalog and BigSpa for both pointer/alias and dataflow analyses on Linux kernel. All experiments done for BigSpa and BigDatalog were conducted in the same cluster with the same Spark installation. Figure 18 shows the performance results under different cluster configurations. The X axis denotes the number of nodes and Y axis indicates the wall-clock execution time.

As for dataflow analysis (Figure 18(a)), BigSpa outperforms BigDatalog significantly. The number of iterations BigSpa needs is much smaller than that of BigDatalog. A large amount of time is taken by BigDatalog for convergence. With regard to the pointer/alias analysis, BigSpa outperforms BigDatalog with all different number of nodes. Note that Figure 18(b) actually shows the performance results for a simplified pointer/alias analysis. Due to implementation issues, BigDatalog cannot support fully-precise pointer analysis. Runtime errors occurred when compiling rules of multiple consecutive joins, e.g., $M ::= \overline{d}\ V\ d$. Reluctantly, we chose a simplified pointer analysis by removing certain rules for the sake of fair comparisons.

Comparison with Distributed Call Graph Analysis: We also compared with another distributed actor model-based analysis tool [20]. We exert ourselves to deploy BigSpa on Microsoft Azure cluster, and implemented a font-end for the inclusion-based Variable Type Analysis [26] to generate the propagation graph, which was fed into BigSpa to compute the CFL-reachability results. Hence, the time cost spent by BigSpa comes from two parts, frontend for propagation graph generation and backend for reachability computation. We conducted all the comparison experiments on a Microsoft Azure A3 cluster with 1 master and 4 workers, each equipped with 4 CPU cores and 7 GB memory. Table 7 shows the performance results for three real programs chosen by [20]. Column 2 (“frontend”) and 3 (“backend”) indicate the time cost by generating propagation graph and computing reachability, respectively.

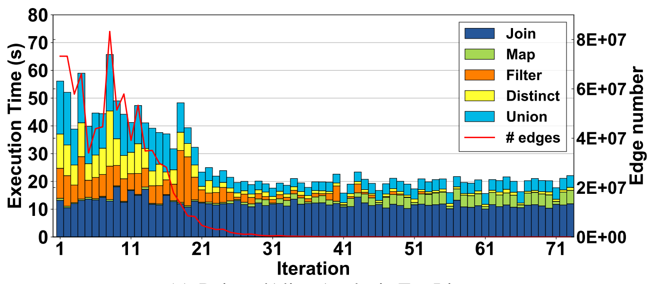
TABLE 7: Performance Comparison with Distributed Call Graph Analysis

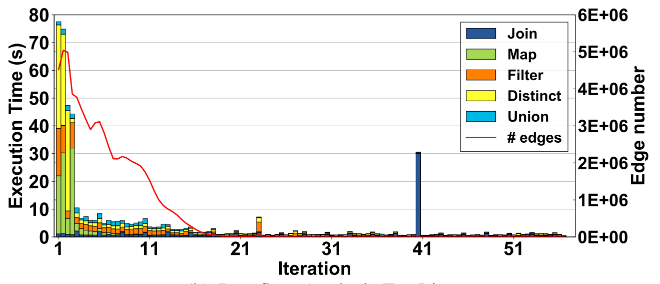
	BigSpa			[20]
Subject	frontend	backend	total	total
Azure-PW	99s	59s	158s	238s
ShareX	49s	49s	98s	190s
ILSpy	97s	121s	218s	1063s

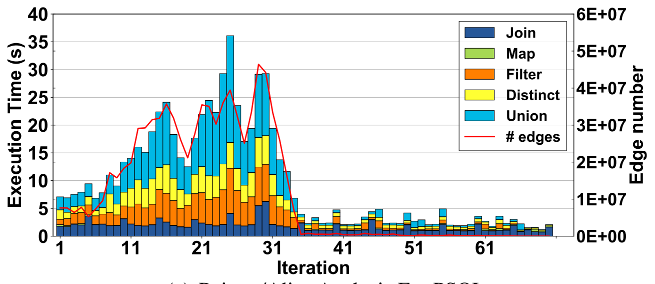
By comparing the total time spent, BigSpa outperforms Distributed Call Graph Analysis[20] especially on large graphs (e.g., more than 4 times faster on ILSpy).

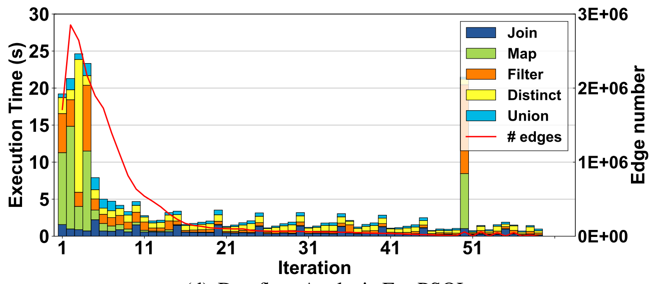
7.2 Evaluation of Online Incremental System

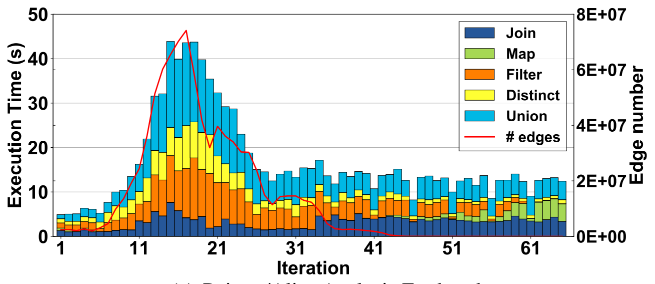
In this subsection, we evaluate the performance of the online incremental static analysis system, including performance analysis

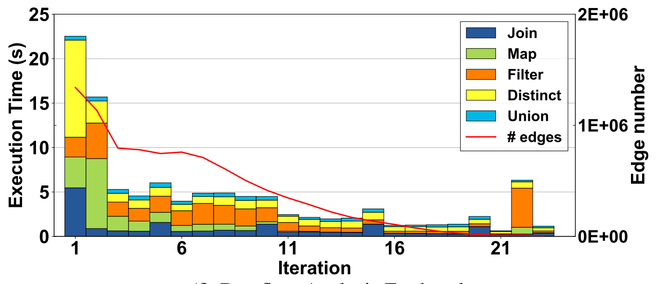

(a) Pointer/Alias Analysis For Linux

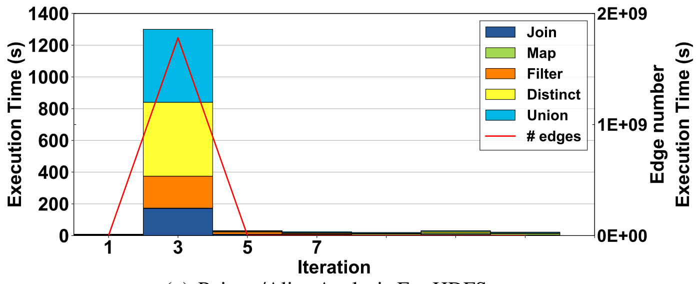

(b) Dataflow Analysis For Linux


(c) Pointer/Alias Analysis For PSQL


(d) Dataflow Analysis For PSQL


(e) Pointer/Alias Analysis For httpd


(f) Dataflow Analysis For httpd


(g) Pointer/Alias Analysis For HDFS

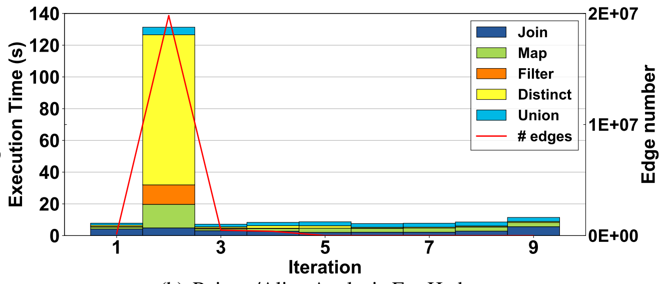
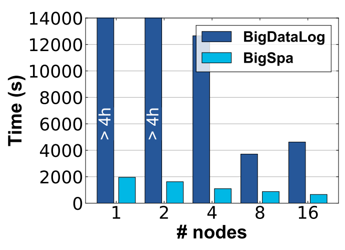

(h) Pointer/Alias Analysis For Hadoop

Fig. 17: Time Cost Distribution. The top and bottom X axes denote the wall-clock execution time and number of newly added edges, respectively. The Y axis indicates different iterations.


(a) Dataflow Analysis

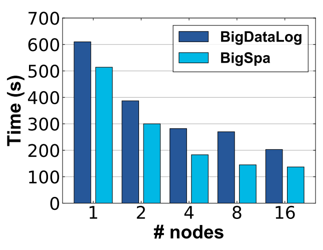

(b) Pointer/Alias Analysis

Fig. 18: Performance Comparisons between BigSpa and BigDatalog on Linux

and comparison with serial incremental analysis on incremental analysis data sets.

7.2.1 *Performance Analysis*

We chose the largest Linux and the smallest httpd as the subjects for this experiment. We produced the program graphs for pointer/alias analysis and data flow analysis, and generated updating batches in two ways: (1) randomly divide them into multiple batches of edges, each with 1000 edges; (2) add and delete nodes based on commits from the realistic version control repositories. We refer to these two experiments as uniform update and commit-based update in the following. The uniform update reflects the process of building large graphs from scratch with medium-scale batch updates, and the latter reflects the code update process in real world. For the commit-based update, we collected tens of thousands of commit records from the GitHub repositories of Linux and httpd respectively and remove the commits that do not involve code changes. Some of these commits only modify a few lines of code, while some others can cause thousands of edges to be modified. The details about the commits can be found in Table 8.

The mini-batches corresponding to both scenarios were then input into the incremental system. To ensure the computation is done in real time, tasks with processing time exceeding the threshold would be interrupted. As can be seen from Table 9 and Table 10, the system has significantly different performance on different tasks. The Response Time column lists the average response time and several upper quantiles. For example, ”0.52” in the first row of Table 9 means that 90% of the updates were completed in 0.52 seconds.

TABLE 8: Information of Commits Collected from GitHub Repositories

Analysis	Subject	# Commits	Proportion of commits with different number of modified edges (%)				
			0-100	100-1000	1000-1E4	1E4-1E5	>1E5
Pointer/Alias	Linux	38285	44.19	38.71	15.04	1.98	0.08
	httpd	7609	63.13	17.68	11.02	6.37	1.8
Dataflow	Linux	38285	66.38	22.21	9.61	1.68	0.12
	httpd	7609	73.9	17.94	7.02	1.05	0.09

TABLE 9: Performance of Online Incremental System in the Uniform Update Experiments

Analysis	Subject	Response Time (s)					Task Interruption Ratio (%)	Throughput (#edges/s)
		Average	90%	95%	99%	99.9%		
Pointer/Alias	Linux	0.31	0.52	0.90	3.33	13.56	0.00	3207
	httpd	6.84	29.04	-	-	-	7.84	147
Dataflow	Linux	0.26	0.39	0.91	4.31	10.71	0.00	3915
	httpd	0.29	0.64	1.36	3.53	6.43	0.00	3441

TABLE 10: Performance of Online Incremental System in the Commit-based Update Experiments

Analysis	Subject	Response Time (s)					Task Interruption Ratio (%)	Throughput (#edges/s)
		Average	90%	95%	99%	99.9%		
Pointer/Alias	Linux	0.06	0.1	0.15	0.5	4.7	0.00	19935
	httpd	0.93	0.77	3.49	21.47	-	0.2	10079
Dataflow	Linux	0.19	0.35	0.73	2.84	13.5	0.00	5507
	httpd	0.13	0.26	0.54	1.59	9.73	0.00	9721

The system shows good performance for pointer/alias analysis of Linux dataset, and for dataflow analysis of Linux and httpd dataset. In these three tasks, thousands of edges are processed per second on average and all the tasks can be finished within 15 seconds. However, in other tasks, the throughput is significantly reduced and timeout interruptions occur.

By monitoring the execution process of the tasks and analyzing the structure of the input graphs, we found that

1) The tasks with calculation time at millisecond level generally have fewer edges to update, and are calculated in the single-machine multi-thread mode.
2) In those tasks completed in seconds, the number of updated edges is larger, but still less than 100,000. In this case the system can still finish the calculation by switching to a distributed computing mode.
3) Overtime tasks often have super-large-scale updates (with hundreds of thousands, or even millions of edges to update) and ”hot vertices” (nodes that are connected to a large number of edges).
4) In general, the system has higher throughput and lower processing latency in the commit-based experiment. This is because in large projects, most of the commits only trigger small-scale code updates, and only a very small number of updates will modify more than 100,000 edges.

In the above experiments, the system shows poor performance when there are a lot of ”hot vertices” and tightly connected vertex clusters in the input mini-batches, as is the case of pointer/alias analysis on httpd. Updating these mini-batches is similar to updating a full graph $(O(|N|^2))$, In contrast, on the Linux dataset, which is the largest one but has a more uniform distribution of edges, the incremental system performs excellently in both kinds of analyses. Therefore, it can be concluded that the online system is suitable for analyzing program graphs with uniform distribution of the transmission edges, and the performance would be affected when performing incremental analysis of program graphs with skew-distributed transmission edges.

From the application scenario, it is not realistic to lower the time cost of precise analysis of large-scale codebases to second level. Because of that, a timeout interrupt mechanism is employed in our implementation to return only part of analysis results in time. If the users want to get a complete and accurate analysis result, they can use the offline batch analysis system for supplementary calculations after the incremental update is over.

7.2.2 *Comparison with Serial Incremental Analysis*

For comparison, in this subsection, the same batches were input into a serial incremental analysis system in a serial manner.

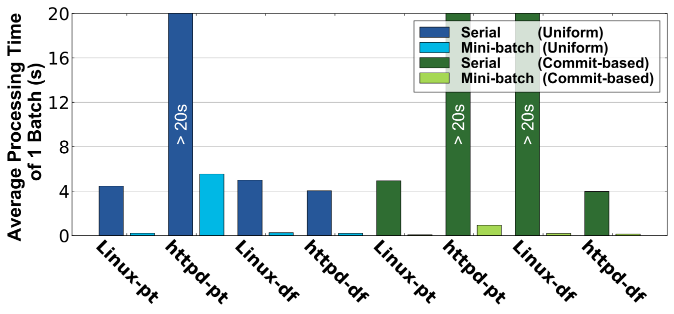


Fig. 19: Comparison with a Serial Incremental Analysis Implementation

The results of the comparison experiment are illustrated in Figure 19. The horizontal axis indicates the software and analysis type, and the vertical axis indicates the average processing time of one batch in seconds. As we can see, the average response time of our mini-batch incremental system is obviously lower (less than one-twentieth) than that of serial system. That is because there are many iterations in the process of incremental analysis, and serial iteration over multiple edges would result in a huge waste of computing resources. Besides, in the serial system, the amount of data to be processed per round is relatively small, therefore the memory can hardly be fully utilized. The above results show that the method of distributed mini-batch processing can effectively handle the incremental updates of program static analysis. Moreover, in practice, it is more common to update the codebases in mini-batches, which means the mini-batch update method is more suitable for real-world application scenarios.

8 Related Work

Static Bug Detection Static analysis is widely used to detect various software defects and security vulnerabilities [1], [2], [3].

Engler et al. [3] took advantage of the simple pattern-based analysis to discover a variety of bugs in Linux kernel. In addition, a series of commercial tools, e.g., Coverity[27], CodeSonar[28], KlocWork[29], are also developed and widely used in industries. Most of these checkers are simply based on patterns/rules or intra-procedural analysis which has relatively low computation complexity, achieving good scalability. However, pattern-based or intra-procedural analysis suffers from severely low accuracy due to the lack of considering adequately rich/complete semantics information. Some empirical studies showed that the false positive rates reach 30%-100%[4], leading to pretty poor applicability[5]. Differently, this work focuses on the sophisticated inter-procedural analysis. By developing a scalable inter-procedural analysis engine, we are able to perform precise sophisticated analysis on large-scale modern software.

CFL Reachability-based Static Analysis CFL-reachability is originally proposed by Yannakakis[15] for Datalog query evaluation. Reps[10] later adopted CFL-reachability to formulate a series of sophisticated static analysis. A variety of static analyses [8], [9], [16], [30] can be viewed as the instances of CFL-reachability problem, e.g., program slicing, inter-procedural dataflow analysis, pointer and alias analysis, shape analysis, specification inference, and information flow analysis. The worst case time complexity for solving CFL-reachability problems is O(n^3), which is called cubic bottleneck[31]. As a result, it is really tough to perform highly precise analysis on large-scale software. Researchers have proposed various optimizations to enhance the analysis efficiency, including compositional analysis[32], sparse representation-based analysis[6], and demand-driven analysis[9] etc. The above work is mainly targeted at the optimized sequential and memory-based algorithm whose scalability is greatly limited.

Incremental Program Analysis For efficient analysis of frequently changing codebases, researchers have also proposed many incremental analysis approaches. Reviser [33] checks the modified version of a code to find the changes of its control flow graph, and then propagates and updates the analysis results. Souter [34] and Ismail [35] employ specialized algorithms for incremental call graph reconstructions in IDE. IncA [36] is a domain-specific language for the definition of efficient incremental program analyses, which functions in a declarative way over AST representations of programs. IncA also includes optimizations for cache reducing and change propagation pruning in order to work on larger programs. Lu et al. [37] combine points-to analysis with graph reachability. They provide a trace-based incremental mechanism that precisely identifies and recomputes affected paths after a program changes. The existing algorithms usually suffer from the lack of good scalability and generality, while our incremental analysis system can incrementalize analysis of very large codebases with distributed computing.

Parallel and Distributed Static Analysis In order to improve the efficiency of program analysis, researchers have developed parallel algorithms. For example, Mendez-Lojo et al. [38] proposed a parallel points-to analysis based on constraint graph rewriting. Su et al. [39] developed a CFL-based parallel pointer analysis algorithm where they leveraged data sharing and query scheduling to avoid redundant graph traversals. Graspan[7] is a single-machine disk-based inter-procedural analysis system using edge-pair centric computation model. It solved the scalability problem to some extent, but still suffers from the limited scalability due to the resource limitation of a single machine. Rodriguez et al. [40] proposed an actor model-based parallel algorithm for dataflow analysis. Moreover, Garbervetsky et al. [20] recently built a distributed program analysis framework on the basis of actor model and implemented call-graph analysis. Albarghouthi et al. [41] parallelized the top-down interprocedural analysis based on MapReduce paradigm. Different from BigSpa, they focused on the demand-driven analysis and only implemented a parallel version running on a multi-core machine. Google also implemented the distributed static analyses to analyze their codebase [42]. But due to the challenge of performing interprocedural analysis on such large-scale code, only relatively simple (intra-procedural) analyses are considered. In addition, Facebook proposed their tool INFER [43] based on bi-abduction to check the memory safety properties of C code. Similarly, for the sake of scalability, INFER only supports the interprocedural analysis within each compilation unit. Blaß et al. [44] presented a parallel approach for fixpoint-based dataflow analysis on a GPU. Although the method performs efficiently, it requires extra accelerators and can hardly scale out. Although the above studies solved the scalability problem to some extent, they still have several drawbacks. Firstly, the above work usually did quite specific optimization to certain particular analysis. They are not applicable to the general program analysis. Secondly, some actor model based algorithms took advantage of locks to implement synchronization. They are not scalable enough due to the big synchronization overheads and large amount of message passing. Finally, most existing parallel and distributed algorithms entirely rely on memory for computation. As stated in [45], the memory would be the major bottleneck for scaling analysis to large programs. It is too challenging to complete precise sophisticated analysis with huge memory consumption. We have also proposed the parallel offline batch interprocedural static program analytic algorithms in the conference version[46] of this paper.

9 Conclusion and Future Work

In this paper, we propose an efficient, distributed interprocedural static analysis engine called BigSpa. We improve the idea of transforming the static analysis problem into a big data processing problem and devise a particular data-parallel algorithm. Based on that, we develop the scalable BigSpa system that can handle both offline batch and online incremental static analysis of programs. Specific optimizations for data structure and processing flow are also designed to ensure high overall efficiency. By utilizing BigSpa, we readily accomplished a series of precise static analyses for large-scale modern software.

In the future, we plan to eliminate redundancy in data storage. Also, we want to explore how to set proper thresholds in the system by using machine learning techniques.

10 Acknowledgments

This work is funded in part by Jiangsu Province Industry Support Program (BK20170651, BK20191247, BE2017155), China NSF Grants (No.61802168, 61702254, 61932021, U1811461), National Key R&D Program of China (2019YFC1711000) and Collaborative Innovation Center of Novel Software Technology and Industrialization.

References

[1] "The findbugs Java static checker," http://findbugs.sourceforge.net/.

[2] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: A tool for finding copy-paste and related bugs in operating system code," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04, pp. 20–20.

[3] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, ser. SOSP '01. ACM, 2001, pp. 57–72.

[4] T. Kremenek and D. Engler, "Z-ranking: Using statistical analysis to counter the impact of static analysis approximations," in *Proceedings of the 10th International Conference on Static Analysis*, ser. SAS'03. Springer-Verlag, 2003, pp. 295–315.

[5] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, pp. 672–681.

[6] J. Whaley and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, ser. PLDI '04. ACM, 2004, pp. 131–144.

[7] K. Wang, A. Hussain, Z. Zuo, G. Xu, and A. Amiri Sani, "Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code," in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17, pp. 389–404.

[8] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95. ACM, 1995, pp. 49–61.

[9] M. Sridharan and R. Bodík, "Refinement-based context-sensitive points-to analysis for java," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '06. ACM, 2006, pp. 387–400.

[10] T. Reps, "Program analysis via graph reachability," in *Proceedings of the 1997 International Symposium on Logic Programming*, ser. ILPS '97. MIT Press, 1997, pp. 5–19.

[11] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[12] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. USENIX Association, 2014, pp. 599–613.

[13] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. USENIX Association, 2012, pp. 31–46.

[14] K. Wang, Z. Zuo, J. Thorpe, T. Q. Nguyen, and G. H. Xu, "Rstream: Marrying relational algebra with streaming for efficient graph mining on a single machine," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*. Carlsbad, CA: USENIX Association, 2018, pp. 763–782. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/wang

[15] M. Yannakakis, "Graph-theoretic methods in database theory," in *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, ser. PODS '90. ACM, 1990, pp. 230–242.

[16] X. Zheng and R. Rugina, "Demand-driven alias analysis for c," in *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '08. ACM, 2008, pp. 197–208.

[17] B. Jurkowiak, C. M. Li, and G. Utard, "Parallelizing satz using dynamic workload balancing," in *Proceedings of Workshop on Theory and Applications of Satisfiability Testing*. Elsevier Science Publishers, 2001, pp. 205–211.

[18] Y. Hamadi and C. M. Wintersteiger, "Seven challenges in parallel sat solving," in *Proceedings of the 26th AAAI Conference on Artificial Intelligence*, ser. AAAI'12. AAAI Press, 2012, pp. 2120–2125. [Online]. Available: http://dl.acm.org/citation.cfm?id=2900929.2901028

[19] D. Melski and T. W. Reps, "Interconveritibility of set constraints and context-free language reachability," in *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 1997, pp. 74–89.

[20] D. Garbervetsky, E. Zoppi, and B. Livshits, "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*, ser. ESEC/FSE 2017. ACM, 2017, pp. 442–453.

[21] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo, "Big data analytics with datalog queries on spark," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. ACM, 2016, pp. 1135–1149.

[22] Y. Zhao, G. Chen, C. Liao, and X. Shen, "Towards Ontology-Based Program Analysis," in *Proceedings of the 30th European Conference on Object-Oriented Programming*, vol. 56, Dagstuhl, Germany, 2016, pp. 26:1–26:25. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2016/6120

[23] M. Sharir and A. Pnueli, "Two approaches to interprocedural data flow analysis," in *Program Flow Analysis: Theory and Applications*, S. Muchnick and N. Jones, Eds., 1981, pp. 189–234.

[24] Z. Zuo, J. Thorpe, Y. Wang, Q. Pan, S. Lu, K. Wang, H. Xu, L. Wang, and X. Li, "Grapple: A graph system for static finite-state property checking of large-scale systems code," in *Proceedings of the 14th European Conference on Computer Systems*, ser. EuroSys '19. ACM, 2019.

[25] F. McSherry, M. Isard, and D. G. Murray, "Scalability! but at what cost?" in *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, ser. HOTOS'15. USENIX Association, 2015.

[26] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin, "Practical virtual method call resolution for java," in *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '00. ACM, 2000, pp. 264–280. [Online]. Available: http://doi.acm.org/10.1145/353171.353189

[27] "The coverity code checker," http://www.coverity.com/.

[28] "The grammatech codesonar static checker," https://www.grammatech.com/products/codesonar.

[29] "The klocwork checker," https://www.klocwork.com/products-services/klocwork.

[30] C. Cai, Q. Zhang, Z. Zuo, K. Nguyen, G. Xu, and Z. Su, "Calling-to-reference context translation via constraint-guided cfl-reachability," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. ACM, pp. 196–210. [Online]. Available: http://doi.acm.org/10.1145/3192366.3192378

[31] N. Heintze and D. McAllester, "On the cubic bottleneck in subtyping and flow analysis," in *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science*, ser. LICS '97. IEEE Computer Society, 1997, pp. 342–.

[32] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. D. Tetali, "Compositional may-must program analysis: Unleashing the power of alternation," in *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL'10. ACM, pp. 43–56.

[33] S. Arzt and E. Bodden, "Reviser: efficiently updating ide-/ifds-based data-flow analyses in response to incremental program changes," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 288–298.

[34] A. L. Souter and L. L. Pollock, "Incremental call graph reanalysis for object-oriented software maintenance," in *Proceedings of International Conference on Software Maintenance*. IEEE, 2001, pp. 682–691.

[35] U. Ismail, "Incremental call graph construction for the eclipse ide," *University of Waterloo Technical Report*, 2009.

[36] T. Szabó, S. Erdweg, and M. Voelter, "Inca: A dsl for the definition of incremental program analyses," in *Proceedings of the 31st International Conference on Automated Software Engineering*. IEEE, 2016, pp. 320–331.

[37] Y. Lu, L. Shang, X. Xie, and J. Xue, "An incremental points-to analysis with cfl-reachability," in *Proceedings of International Conference on Compiler Construction*. Springer, 2013, pp. 61–81.

[38] M. Méndez-Lojo, A. Mathew, and K. Pingali, "Parallel inclusion-based points-to analysis," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '10. ACM, 2010, pp. 428–443.

[39] Y. Su, D. Ye, and J. Xue, "Parallel pointer analysis with cfl-reachability," in *Proceedings of the 43rd International Conference on Parallel Processing*, 2014, pp. 451–460.

[40] J. Rodriguez and O. Lhoták, "Actor-based parallel dataflow analysis," in *Proceedings of the 20th International Conference on Compiler Construction*, ser. CC'11/ETAPS'11, 2011, pp. 179–197.

[41] A. Albarghouthi, R. Kumar, A. V. Nori, and S. K. Rajamani, "Parallelizing top-down interprocedural analyses," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. ACM, 2012, pp. 217–228. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254091

[42] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at google," *Communications of the ACM*, vol. 61, no. 4, pp. 58–66, Mar. 2018. [Online]. Available: http://doi.acm.org/10.1145/3188720

[43] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez, "Moving fast with software verification," in *NASA Formal Methods*, K. Havelund, G. Holzmann, and R. Joshi, Eds., Cham, 2015, pp. 3–11.

[44] T. Blaß and M. Philippsen, "Gpu-accelerated fixpoint algorithms for faster compiler analyses," 2019, pp. 122–134.

[45] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins, "An overview of the saturn project," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '07. ACM, 2007, pp. 43–48. [Online]. Available: http://doi.acm.org/10.1145/1251535.1251543

[46] Z. Zuo, R. Gu, X. Jiang, Z. Wang, Y. Huang, L. Wang, and X. Li, "Bigspa: An efficient interprocedural static analysis engine in the cloud," in *Proceedings of the 2019 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2019, pp. 771–780.

[47] Q. Zhang, X. Xiao, C. Zhang, H. Yuan, and Z. Su, "Efficient subcubic alias analysis for c," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, 2014.

Rong Gu is an associate researcher at State Key Laboratory for Novel Software Technology, Nanjing University, China. Dr. Gu received the Ph.D. degree in computer science from Nanjing University in December 2016. His research interests include distributed storage, parallel computing, and distributed machine learning.

Zhiqiang Zuo received his PhD from National University of Singapore in 2015. He is now an assistant professor in the State Key Laboratory of Novel Software Technology at Nanjing University, China. His research interests span programming languages, software engineering, and systems.

Xi Jiang received her BS degree in Nanjing University of Science and Technology, China. She is currently working towards the MS degree in Nanjing University. Her research interests include distributed storage system and parallel algorithms.

Han Yin received his BS degree in Nanjing University, China. He is currently working towards the MS degree in Nanjing University. His research interests include distributed storage system and parallel algorithms.

Zhaokang Wang received his BS degree in Nanjing University in 2013. He is currently working towards the PhD degree in Nanjing University. His research interests include distributed graph algorithms and graph processing systems.

Linzhang Wang received the PhD degree from Nanjing University, China in 2005. He is currently a full professor in the State Key Laboratory of Novel Software Technology at Nanjing University, China. His research interests include software engineering and software security.

Xuandong Li received the MS and PhD degrees from Nanjing University, China in 1991 and 1994, respectively. He is a full professor in the State Key Laboratory of Novel Software Technology at Nanjing University, China. His research interests include formal support for program design and analysis, software testing and verification.

Yihua Huang is a professor in computer science department and State Key Laboratory for Novel Software Technology, Nanjing University, China. He received his Ph.D. degrees in computer science from Nanjing University. His main research interests include parallel and distributed computing and big data parallel processing.

APPENDIX

.1 Grammar for Pointer/Alias Analysis

We adopt the following grammar (Rules 13-16) described in [16] as our pointer/alias grammar. The normalized version [47] is shown as Rules 17-28.

Original grammar:

$$\textit{Memory aliases} \quad M ::= \overline{d}\, V\, d \tag{13}$$

$$\textit{Value aliases} \quad V ::= \overline{F}\, M?\, F \tag{14}$$

$$\textit{Flows of values} \quad F ::= (a\, M?)^* \tag{15}$$

$$\overline{F} ::= (M?\, \overline{a})^* \tag{16}$$

Normalized grammar:

$$M ::= DV\, d \tag{17}$$

$$DV ::= \overline{d}\, V \tag{18}$$

$$V ::= MAM\, AMs \tag{19}$$

$$MAM ::= MAs\, Mq \tag{20}$$

$$Mq ::= \epsilon \tag{21}$$

$$Mq ::= M \tag{22}$$

$$MAs ::= \epsilon \tag{23}$$

$$MAs ::= MAs\, MA \tag{24}$$

$$MA ::= Mq\, \overline{a} \tag{25}$$

$$AMs ::= \epsilon \tag{26}$$

$$AMs ::= AMs\, AM \tag{27}$$

$$AM ::= a\, Mq \tag{28}$$

There are four terminal symbols: a, $\overline{a}$, d, and $\overline{d}$. a and d represent *address-of* and *dereference* relation, while $\overline{a}$ and $\overline{d}$ represent the inverse edge of a and d, respectively. All other capitalized symbols are non-terminals. As seen in normalized grammar, we introduced several temporary non-terminals, making sure that each production has at most two symbols at the right hand side.

For dataflow analysis, we used the original grammar shown as Rule 29

$$n ::= n\, e \tag{29}$$

.2 Proof of Correctness of Triangle Counting Method

This section proves that we can correctly determine the reachability relationship by querying triangle counts.

Let the graph $\mathcal{G} = (V, E, \Sigma)$ and its transitive closure is TC. $\forall a, b \in V$, $\forall \ell \in \Sigma$, $tri\_count(a \xrightarrow{\ell} b)$ is the triangle count of edge $a \xrightarrow{\ell} b$. $Query((a, b, \ell) \in TC)$ indicates whether the reachability relationship (a, b, ℓ) already is already in the transitive closure TC by querying the triangle counts. The relationship between triangle count and TC can be represented as:

$$Query((a,b,\ell) \in TC) = \begin{cases} true, & if \quad tri\_count(a \xrightarrow{\ell} b) > 0 \\ false, & if \quad tri\_count(a \xrightarrow{\ell} b) = 0 \end{cases} \tag{30}$$

Formular 30 means that b is ℓ-reachable from a only if $tri\_count(a \xrightarrow{\ell} b) > 0$. Note that the reachability relationship here only refers to indirect reachability, and direct reachability is not calculated by the triangle count.

Theorem 1. In calculation of transitive closure, the reachability relationship can be correctly determined by querying the triangle count.

Let the state where the transitive closure is empty be the initial state S_0. Suppose there are two intermediate states S_k and S_{k+1}. The transition between states is done by updating the reachability relationship, which is expressed as $S_k \xrightarrow{\Delta} Sk + 1$. The update of reachability relationship Δ is the addition and deletion of reachability relationship (a, b, ℓ).

Use mathematical induction to prove:

First, in the initial state S_0, $tri\_count(a \xrightarrow{\ell} b) = 0$, then $Query((a, b, \ell) \in TC) = false$. This is consistent with the fact that the transitive closure is empty.

Second, assuming the conclusion holds in state S_k, we prove that it still holds in state S_{k+1}:

In state S_k, assuming $tri\_count(a \xrightarrow{\ell} b) = \alpha$, there are two cases according to the value of α:

1) $\alpha > 0$. $Query((a, b, \ell) \in TC) = true$, which means the reachability relationship (a, b, ℓ) exists. If $\alpha = 1$, it means (a, b, ℓ) is generated by one two-hop path. Otherwise it is generated by multiple two-hop paths.
2) $\alpha = 0$. $Query((a, b, \ell) \in TC) = false$, which means the reachability relationship (a, b, ℓ) does not exist.

When the state is transformed into S_{k+1}, the tri_count and TC change as follows according to the different values of $Delta$:

1) If $Delta$ is an addition. In this case, $tri\_count(a \xrightarrow{\ell} b) = \alpha + 1$, and $Query((a, b, \ell) \in TC)$ must be true. During the calculation of transitive closure, if $\alpha > 0$, we know that (a, b, ℓ) already exists and the updata $Delta$ will not trigger the change of other reachability relations and itself. If $\alpha = 0$, a new reachability relation (a, b, ℓ) is generated and it may trigger the generation of other reachability relations by looking at the other reachability relationship between a, b and then performing label matching. The new reachability relations will be added to the update of next round of iteration Δ_{next}. In both cases of $\alpha > 0$ and $\alpha = 0$, the reachability relationship (a, b, ℓ) exists in the transitive closure, so the result returned by $Query((a, b, \ell) \in TC)$ is correct.
2) If $Delta$ is an deletion. $tri\_count(a \xrightarrow{\ell} b) = \alpha - 1$. This can happend only when $\alpha > 0$. If $\alpha > 1$, $tri\_count(a \xrightarrow{\ell} b) > 0$ and $Query((a, b, \ell) \in TC) = true$; If $\alpha = 1$, $tri\_count(a \xrightarrow{\ell} b) = 0$ and $Query((a, b, \ell) \in TC) = false$. During the calculation of transitive closure, if $\alpha > 1$, we know that (a, b, ℓ) is generated by multiple paths and it is not affected by the deletion in $Delta$. Therefore $Query((a, b, \ell) \in TC)$ should be true; If $\alpha = 1$, (a, b, ℓ) is generated by only one path and the deletion should make (a, b, ℓ) be completely deleted, which means $Query((a, b, \ell) \in TC)$ should be false. Similar as in (1), A completely deleted reachable relationship may iteratively trigger the deletion of other reachable relationships, and the newly deleted reachable relationship obtained by calculation is added to Δ_{next}. In the two cases of $\alpha > 1$ and $\alpha = 1$, the result returned by $Query((a, b, \ell) \in TC)$ is correct.

Therefore, in both cases, the conclusion still holds in the S_{k+1} state.

In summary, Theorem 1 holds, which proves that in the calculation of transitive closure, the reachability can be correctly determined by querying the triangle count. check update.