BigDataflow: A Distributed Interprocedural Dataflow Analysis Framework

Zewen Sun*
Nanjing University, China
sunzew@smail.nju.edu.cn

Duanchen Xu*
Nanjing University, China
mf1933108@smail.nju.edu.cn

Yiyu Zhang*
Nanjing University, China
zhangyy0721@smail.nju.edu.cn

Yun Qi*
Nanjing University, China
mf20330058@smail.nju.edu.cn

Yueyang Wang*
Nanjing University, China
181860105@smail.nju.edu.cn

Zhiqiang Zuo*†
Nanjing University, China
zqzuo@nju.edu.cn

Zhaokang Wang*
Nanjing University, China
wang.zk@foxmail.com

Yue Li*
Nanjing University, China
yueli@nju.edu.cn

Xuandong Li*
Nanjing University, China
lxd@nju.edu.cn

Qingda Lu
Alibaba Group, United States
qingda.lu@alibaba-inc.com

Wenwen Peng
Alibaba Group, China
wenwen.pww@alibaba-inc.com

Shengjian Guo
Baidu Research, United States
guosj@vt.edu

ABSTRACT

Apart from forming the backbone of compiler optimization, static dataflow analysis has been widely applied in a vast variety of applications, such as bug detection, privacy analysis, program comprehension, etc. Despite its importance, performing interprocedural dataflow analysis on large-scale programs is well known to be challenging. In this paper, we propose a novel distributed analysis framework supporting the general interprocedural dataflow analysis. Inspired by large-scale graph processing, we devise a dedicated distributed worklist algorithm tailored for interprocedural dataflow analysis. We implement the algorithm and develop a distributed framework called BigDataflow running on a large-scale cluster. The experimental results validate the promising performance of BigDataflow – it can finish analyzing the program of millions lines of code in minutes. Compared with the state-of-the-art, BigDataflow achieves much more analysis efficiency.

CCS CONCEPTS

• **Software and its engineering** → **General programming languages**; • **Theory of computation** → *Program analysis*; • **Computing methodologies** → Distributed algorithms.

KEYWORDS

interprocedural dataflow analysis, distributed computing

*Also with State Key Laboratory for Novel Software Technology at Nanjing University.
†Corresponding author.

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3616348

ACM Reference Format:
Zewen Sun, Duanchen Xu, Yiyu Zhang, Yun Qi, Yueyang Wang, Zhiqiang Zuo, Zhaokang Wang, Yue Li, Xuandong Li, Qingda Lu, Wenwen Peng, and Shengjian Guo. 2023. BigDataflow: A Distributed Interprocedural Dataflow Analysis Framework. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616348

1 INTRODUCTION

Dataflow analysis is a technique for statically gathering program information at program points along the program's control flow. Besides forming the backbone of compiler optimization, it has been adopted in many other significant application areas, including bug detection [37, 45], security vulnerability discovery [22], privacy analysis [3], program testing/debugging [43, 49], etc. In a dataflow analysis, a separate dataflow fact is maintained at each program point under the control flow graph (CFG) representation. Based on the effect of each statement, a transfer function is applied to transform the dataflow fact accordingly along the CFG. The transformation process is performed iteratively via a worklist algorithm until a fixed point is reached [29], meaning that all the dataflow facts are unchanged anymore.

Challenges. Despite its importance, performing interprocedural dataflow analysis on large-scale systems code is well known to be challenging. First, as modern real-world programs are usually of large scale (like million lines of code), maintaining solutions at all program points with limited memory can hardly be scalable. Even worse, for certain analysis, the dataflow solution maintained at each point itself is highly space-intensive. Although prior work attempts to adopt sparse representations [7, 15, 33, 45], the huge memory consumption still severely limits the scalability. As evidenced by recent studies [41, 53], the analysis over sparse value-flow graph can easily exceed hundreds of Gigabytes, showing the memory consumption a factual bottleneck. Second, the computation of flow-sensitive analysis requires updating the dataflow fact with respect

to each statement along the CFG by performing the transfer function. The process is highly computation-intensive because: (1) the amount of transfer function executed is at least linear in the number of program statements, which is large-scale given the modern large-size software under analysis; (2) the computation of each transfer function is perhaps expensive as well. For instance, in the flow-sensitive pointer/alias analysis, the dataflow fact at each program point should capture the alias information among all the variables in the entire program. Updating variable relations by each transfer function consumes high CPU cycles.

State-of-the-Art. To accelerate interprocedural dataflow analysis, a few attempts to distribute/parallelize the computation have been made. For distributed approaches, Garbervetsky et al. [10] presented a distributed worklist algorithm on the basis of the actor model. However as stated explicitly in their paper, it cannot support the standard dataflow analysis due to the lack of flow ordering between actors. Albarghouthi et al. [2] parallelized the demand-driven top-down analyses based on MapReduce paradigm. They only targeted verification and software model checking without supporting dataflow analysis. BigSpa [14, 51] supports the distributed acceleration for CFL reachability-based analysis [34]. Unfortunately, a lot of dataflow analyses, e.g., cache analysis and numerical analysis, do not belong to this category. Greathouse et al. [13] proposed scalable dataflow analysis. However, they focused on dynamic analysis rather than static analysis. In brief, there exist no distributed systems supporting static dataflow analysis.

As for parallel approaches, Lee and Ryder [23] exploited algorithmic parallelism to accelerate dataflow analysis. Rodriguez et al. [36] proposed an actor model-based parallel algorithm for interprocedural finite distributive subset (IFDS) analysis [35]. Moreover, some researchers [31, 44] also studied parallel algorithms for pointer analysis. Note that the above approaches only support specific analysis rather than the general class of dataflow analyses. More importantly, they rely heavily on memory for computation. There is no doubt that they can rarely scale to large systems such as Linux kernel [1, 53]. Recently, Zuo et al. [53] developed Chianina, a single machine-based analysis framework which can scale general dataflow analysis to millions lines of code. Unfortunately, due to the involvement of disks, it readily takes hours or even days to finish the analysis for large-scale programs. Such inefficiency can hardly meet the requirement of quick analysis response (usually in minutes) in the modern continuous integration and deployment (CI/CD) pipelines [8, 38].

Our Work. With the advent of cloud computing, the large-scale distributed cluster of commodity computers has become prevalent. It not only offers powerful computing capability, but nowadays can be easily accessible by a single developer. Exploiting cloud resources for static analysis would be the promising breakthrough point for achieving both significant scalability and efficiency. However, as mentioned earlier, there exists no distributed system running on a cluster which can support the general dataflow analysis. Adapting the existing parallel algorithms (such as Chianina) to distributed environment is non-trivial. Parallel algorithms only focus on computation on shared memory, which lacks the consideration of partitioning, task dispatching, fault tolerance, and efficient communications between cluster nodes. None of the existing parallel approaches can directly do it without re-designing and re-implementing the system. In this work, we propose a novel system that can leverage large-scale distributed cloud resources to scale and accelerate the general class of interprocedural dataflow analyses. In particular, it only takes minutes to analyze the programs of millions lines of code provided that a cluster of 125 commodity PCs.

Inspired by large-scale graph processing [20, 27, 28], we revisit the traditional worklist algorithm from the perspective of *distributed vertex-centric computation model*, and devise a dedicated distributed worklist algorithm tailored for interprocedural dataflow analysis. We implement the distributed algorithm atop the general distributed graph processing platform (*i.e.*, Apache Giraph [6, 39]) and develop a framework named BigDataflow running on the cloud so as to take full advantage of the modern distributed computing resources. The underlying platform (*i.e.*, Apache Giraph) provides the basic functionalities to support reliable and robust distributed processing, including input partitioning, task dispatching, cross-node communications, and fault tolerance. BigDataflow, as a generic framework, provides several APIs to specify the transfer functions and merge operator similar to other monotone dataflow frameworks [5, 32], thus alleviating the burden of implementing various client analyses. By filling these APIs, users can readily implement a particular dataflow analysis on top of BigDataflow.

Contributions. The contributions are listed as follows:

- We devise an optimized distributed vertex-centric computation model to accelerate static dataflow analysis by leveraging large-scale cloud resources.
- We develop and implement a distributed dataflow analysis framework called BigDataflow running on a real-world cloud, which provides a variety of high-level APIs to easily implement client dataflow analyses.
- We evaluate the performance and scalability of BigDataflow over large-scale real-world software systems (e.g., Firefox and Linux kernel). The experimental results validate the promising performance of BigDataflow—it can finish analyzing the program of millions lines of code in minutes.

Outlines. The rest of the paper is organized as follows. § 2 gives the necessary background of dataflow analysis and distributed graph processing. § 3 presents the distributed worklist algorithms proposed, followed by the implementation details of BigDataflow in § 4. We discuss the programming model provided by our framework to implement various client analyses in § 5. § 6 describes the empirical evaluation of BigDataflow in terms of performance and scalability. We give certain discussions in § 7 and review the related work in § 8. Finally, § 9 concludes.

2 BACKGROUND

2.1 Intraprocedural Dataflow Analysis

Dataflow analysis is a technique for gathering program information with respect to various program points along program flows. A client dataflow analysis can usually be formulated as an instance of the monotone dataflow analysis framework [19, 21], which consists of the analysis domain including operations to copy and merge domain elements, and the transfer functions over domain elements with respect to each type of statement in the control flow graph

(CFG). An iterative worklist algorithm then takes as input an instance of the monotone framework, performs the transfer function for each program statement iteratively along the CFG, and computes a fixed point as the analysis result [18]. Algorithm 1 shows the worklist algorithm for forward analysis in detail.

For each statement k in the CFG, two elements $\mathcal{IN}_k$ and $\mathcal{OUT}_k$ represent the incoming and outgoing dataflow facts, respectively. At each merging point of CFG in which case a node k has multiple predecessors $p \in preds(k)$, the incoming dataflow fact $\mathcal{IN}_k$ of node k is the combination of all the outgoing facts $\mathcal{OUT}_p$ (shown as Line 4) where $\otimes$ indicates the merge operator specified by users which can be meet (for must-analysis) or join (for may-analysis). A transfer function for statement k then takes as input $\mathcal{IN}_k$ and returns the new outgoing fact, as shown by Line 5. The worklist algorithm is conducted along the CFG to update the dataflow elements $\mathcal{IN}_k$ and $\mathcal{OUT}_k$ for each statement in an iterative manner until a fixed point is reached, meaning that all the dataflow facts are unchanged anymore [18].

Algorithm 1: Worklist Algorithm for Forward Analysis

```
1  W ← {all the entry statements of the CFG}
2  repeat
3      remove k from W
4      IN_k ← ⊗_{p∈preds(k)} OUT_p /*merge function*/
5      Temp ← (IN_k \ KILL_k) ∪ GEN_k /*transfer function*/
6      if Temp ≠ OUT_k then
7          OUT_k ← Temp
8          W ← W ∪ succs(k)
9  until W ≡ ∅
```

2.2 Interprocedural Dataflow Analysis

Interprocedural dataflow analysis takes into account the propagation of dataflow values across multiple procedures. Context-sensitive interprocedural analysis distinguishes the distinct calls of a procedure to eliminate the invalid paths, thus achieving high precision. Generally, there exist two dominant approaches to context-sensitive interprocedural analysis, namely the *summary-based (or functional) approach* and the *cloning-based approach* [40].

The summary-based approach commonly constructs a summary (transfer) function for each procedure. At each call site where the procedure is invoked, the analysis computes the effects of the procedure by directly applying the summary function to the specific inputs at the call site. As such, the re-analysis of the procedure body is avoided while enabling context sensitivity. However, it is not possible to construct such (symbolic) summary functions in general. Take the pointer analysis as an example, we can hardly establish a succinct summarization for each procedure since the effects of a procedure are heavily dependent of the alias relations of the inputs at each call site. The evaluation of a summary function on a particular input may not be cheaper than reanalyzing the whole procedure [48]. Another option is the explicit representation, a.k.a. tabulation method or partial transfer functions [30, 48]. Given a finite lattice, it enumerates the summary function as input-output dataflow value pairs for each procedure. The output value of a summary function can be directly exploited when the identical input value is encountered again for the same procedure. However, as a large number of states need to be maintained, this approach usually suffers from huge space consumption.

The alternative of achieving context-sensitivity is a cloning-based approach, where a separate clone of the procedure body is created at each callsite [9, 47]. As such, each procedure is re-analyzed under each calling context, preventing the analysis from propagating dataflow values along invalid paths. In this work, we adopt the cloning-based approach to achieve context-sensitivity. The basic analysis logic of interprocedural analysis is the same as that of intraprocedural analysis shown as Algorithm 1, except that the CFG becomes the interprocedural CFG. More specifically, to construct the interprocedural CFG, the CFG for each function is firstly generated. Based on a pre-computed call graph, the CFG for each function is cloned and incorporated into that of each of its callers by creating assignment edges to connect vertices representing formal parameters and actual arguments. In order to achieve the sweet spot between scalability and precision, we can actually perform cloning only at certain levels, which is theoretically equivalent to the k-CFA call string approach [42].

2.3 Vertex-Centric Graph Processing

With the inception of Pregel system [27], vertex-centric graph processing becomes a hotspot in the large-scale graph processing community [20]. Following Pregel, various algorithmic techniques and systems were proposed, such as asynchronous model (GraphLab [25]), in-memory data parallel model (GraphX [12]). People are able to achieve efficient, scalable, and fault-tolerant graph computing on a large cluster of computers by leveraging these systems.

Algorithm 2: Synchronous Vertex-centric Graph Processing

```
Data: A: the set of active vertices during processing
1  repeat
2      for each vertex k ∈ A do in parallel /*done by system*/
3          Remove k from A /*done by system*/
4          /*perform user-specified logic for each vertex, in particular
             including Gather, Apply and Scatter*/
5          M_k ← Gather(k) /*gather messages or information
             from neighbors*/
6          D_k ← Apply(M_k, k) /*update value of k based on
             gathered information*/
7          ⟨M, A'⟩ ← Scatter(D_k, k) /*activate new vertices
             and/or send out messages*/
8      /*synchronize before next superstep*/
9      Synchronize() /*done by system*/
10     A ← A' /*done by system*/
11 until A ≡ ∅
```

Algorithm 2 gives the pseudo-code of a synchronous vertex-centric processing algorithm. Given an initialized set of active vertices, it conducts an iterative computation where each iteration is termed as a superstep. At each superstep, all the active vertices in $\mathcal{A}$ are processed in a distributed and parallel way across the entire cluster. Over each active vertex k, Gather-Apply-Scatter (a.k.a.,

GAS model) is performed [11]. At first, the messages or information from its neighbors are gathered (Line 5). At the Apply phase, it updates its associated value $\mathcal{D}_k$ according to its current value and the information gathered (Line 6). Based on the newly computed value, it updates the active vertices accordingly, and/or sends necessary messages to its neighbors (Line 7) at the Scatter phase. Before the next superstep, all the messages generated at the current superstep and active vertices are synchronized (Lines 9-10). The whole computation terminates until no active vertex is generated.

Note that vertex-centric graph processing [11, 27, 28] is a programming model for implementing graph processing applications. Users write graph algorithms from the perspective of vertices. They only need to specify the code executed at each vertex, particularly Gather-Apply-Scatter functions (Lines 5-7). The underlying graph processing system is responsible for dividing the input large-scale graph into multiple partitions, loading partitions into different cluster nodes, launching multiple threads/processes to execute user-defined code simultaneously, performing necessary synchronizations, optimizing communication among nodes, maintaining replicas to ensure fault tolerance, etc.

In this work, we take inspiration from vertex-centric graph processing, design and implement a distributed framework BigDataflow tailored to interprocedural dataflow analysis of large-scale code. Similar to the existing general-purpose graph systems, BigDataflow provides user-friendly APIs (*e.g.*, merge and transfer functions) based on which users can readily implement their own client analyses without worrying about scalability. The intrinsic system support under BigDataflow ensures the distributed capability in lifting the sophisticated analysis to large-scale programs.

3 DISTRIBUTED VERTEX-CENTRIC WORKLIST ALGORITHM

Inspired by large-scale graph processing, we revisit the classic worklist algorithm of dataflow analysis (Algorithm 1) from the perspective of vertex-centric computation model (Algorithm 2), and accordingly present our first distributed worklist algorithm, *i.e.*, Algorithm 3 in § 3.1. This algorithm faithfully follows the classic worklist algorithm, and thus it is easy to understand; however, its scalability is also limited under the distributed setting. As a result, in § 3.2, we further propose an optimized algorithm that achieves better performance than Algorithm 3 as demonstrated in § 6.

3.1 Distributed Worklist Algorithm

By directly instantiating Gather-Apply-Scatter interface and other respective data structures in Algorithm 2, we devise the first distributed worklist algorithm for dataflow analysis, which is listed as Algorithm 3.

Our first worklist algorithm takes as input a large interprocedural control flow graph (CFG) or an arbitrary sparse representation [7, 16, 33]. At the beginning, all the entry vertices in the input CFG are added to $\mathcal{W}$ as the initial active vertices (Line 1). During each superstep, the underlying system launches a large number of threads/processes to handle the computation on each vertex in parallel (Line 3). On each vertex k, all the dataflow facts from k's predecessors are firstly gathered (Line 5). This can be implemented by directly invoking the existing APIs provided by pull-based graph

Algorithm 3: Distributed Worklist Algorithm

Data: $\mathcal{W}$: the list of all active vertices during analysis;
$\mathcal{DS}_k : \{\mathcal{OUT}_p \mid p \in preds(k)\}$ a set containing all the dataflow facts of k's predecessors

```
1  W ← {all the entry vertices in CFG}
2  repeat
3      for each CFG vertex k ∈ W do in parallel
4          Remove k from W
5          DS_k ← GatherAll(k)  /*gather all the predecessors' dataflow facts*/
6          IN'_k ← Merge(DS_k)  /*merge*/
7          OUT'_k ← Transfer(IN'_k, k)  /*transfer*/
8          if Propagate(OUT_k, OUT'_k) then /*propagate*/
9              OUT_k ← OUT'_k
10             W' ← W' ∪ succs(k)
11     Synchronize()
12     W ← W'
13 until W ≡ ∅
```

systems (*e.g.*, PowerGraph [11]) or designing a pulling mechanism on top of push-based systems (such as Giraph [6]). Next a *Merge* function takes all the dataflow facts gathered from predecessors (*i.e.*, $\mathcal{DS}_k$) as input, and produces the incoming dataflow fact $\mathcal{IN}'_k$ (Line 6). A transfer function is then performed to generate the new outgoing dataflow fact $\mathcal{OUT}'_k$ (Line 7). After that, we check if the updated dataflow fact $\mathcal{OUT}'_k$ is different from that (*i.e.*, $\mathcal{OUT}_k$) at previous superstep. If so, the propagation is employed to update the dataflow fact as the newly computed value (Line 9). Simultaneously, all of k's successors are activated and put into active list $\mathcal{W}'$ for the next superstep (Line 10).

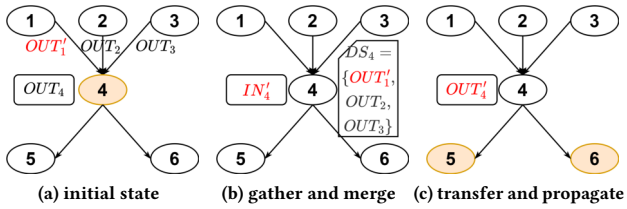


Figure 1: One superstep computation at vertex 4 in Algo. 3.

Example. Figure 1 illustrates the computation procedure at vertex 4 in the above algorithm, where the vertices with yellow background are active. Vertices 1, 2, 3 are the predecessors of 4, and 5, 6 are its successors. Suppose that at the beginning of a certain superstep, the active vertex 4 has its outgoing dataflow fact $\mathcal{OUT}_4$. Predecessor 1 has the newly updated fact $\mathcal{OUT}'_1$, while predecessors 2 and 3 hold the old dataflow facts $\mathcal{OUT}_2$ and $\mathcal{OUT}_3$, respectively (shown as Figure 1a). Firstly, all the predecessors' dataflow facts are gathered as $\mathcal{DS}_4 = \{\mathcal{OUT}'_1, \mathcal{OUT}_2, \mathcal{OUT}_3\}$ and $\mathcal{IN}'_4$ is generated by merging $\mathcal{DS}_4$ shown in Figure 1b. $\mathcal{OUT}'_4$ is computed by performing transfer function on $\mathcal{IN}'_4$. Assuming that $\mathcal{OUT}'_4$ is different from $\mathcal{OUT}_4$, propagation is employed so that all the successors 5 and 6 are marked as active shown as Figure 1c.

Despite that the above algorithm succeeds in leveraging large-scale distributed computing resources to accelerate dataflow analysis, it may still suffer from poor scalability especially when analyzing large-scale programs such as the Linux kernel or Firefox (elaborated shortly in § 6). As shown in Algorithm 3, each vertex has to collect a full set of dataflow facts associated with all its predecessors (*i.e.*, $\mathcal{DS}_k$) for computation. In the worst case, the dataflow fact of each vertex would be made multiple copies each of which is sent to one of its successors. As a result, the total number of dataflow facts held in memory and passed across networks grows exponentially with the size of interprocedural control flow graph under analysis. This number could be super large in practice especially when performing context-sensitive analysis over large-scale programs. Passing/gathering a huge number of expensive dataflow facts not only exhausts the precious memory of a cluster quickly but also increases the burden of network communications, leading to poor scalability. We implemented Algorithm 3 as a prototype named BigDataflow-classic and conducted the empirical evaluation of it. The experimental results discussed shortly in § 6 show that BigDataflow-classic works well for medium-size programs, but quickly runs out of memory on a 500-worker cluster when analyzing large-scale programs, such as the Linux kernel or Firefox. In the following (§ 3.2), we propose an optimized algorithm which addresses the aforementioned limitations by significantly pruning away the data gathered, thus achieving better scalability and performance.

3.2 Optimized Distributed Worklist Algorithm

Algorithm 4: Optimized Distributed Worklist Algorithm

Data: $\mathcal{W}$: the list of all active vertices during analysis;
$\mathcal{M}_k$: $\{\mathcal{OUT}'_p \mid \text{p is a predecessor of k}\}$ a set containing the dataflow facts of k's predecessors which are updated at previous superstep

```
1  W ← {all the entry vertices in CFG}
2  repeat
3      for each CFG vertex k ∈ W do in parallel
4          Remove k from W
5          M_k ← GatherMessages(k)  /*gather dataflow facts of
             the updated predecessors*/
6          IN'_k ← Merge(M_k, IN_k)  /*merge*/
7          OUT'_k ← Transfer(IN'_k, k)  /*transfer*/
8          if Propagate(OUT_k, OUT'_k) then /*propagate*/
9              OUT_k ← OUT'_k
10             foreach successor d of k do
11                 SendMessages(d, OUT'_k)  /*send the updated
                     dataflow facts to successors*/
12                 W' ← W' ∪ {d}
13         IN_k ← IN'_k
14     Synchronize()
15     W ← W'
16 until W ≡ ∅
```

As discussed earlier, each active vertex requires the dataflow facts associated with all its predecessors to complete the computation in the original worklist algorithm (*i.e.*, Line 4 of Algorithm 1 and Line 6 of Algorithm 3). That is why extensive dataflow facts have to be transferred across the cluster network and then merged locally on each vertex, resulting in poor scalability. To tackle the problem, we devise an optimized algorithm which prunes the dataflow facts to be gathered. In particular, instead of gathering the full set of dataflow facts from all the predecessors, only the predecessors' dataflow facts that are newly updated at the previous superstep are passed and merged. Since a significant portion of dataflow facts are not changed at one superstep, the optimized algorithm can thus prune away many unnecessary and memory-consuming dataflow facts to be gathered, greatly reducing the overall message traffic and computation cycles for merging. We will discuss the correctness of such optimization – it produces the same analysis results as the original algorithm, and give the formal proof shortly in §3.3.

We propose an optimized distributed worklist algorithm shown as Algorithm 4. For each active vertex k, only the set of predecessors' dataflow facts which are updated at previous superstep are gathered. This can be achieved via the push-based message passing mechanism. Specifically, each vertex k passively receives the messages passed to it (*i.e.*, $\mathcal{M}_k$) from its predecessors at previous superstep (Line 5). Each message in fact corresponds to a dataflow fact sent from one of the predecessors which is updated at the previous superstep. Subsequently, the dataflow facts $\mathcal{OUT}'_p \in \mathcal{M}_k$ are merged with the incoming dataflow fact of k at last superstep (*i.e.*, $\mathcal{IN}_k$) to generate the new incoming dataflow fact (*i.e.*, $\mathcal{IN}'_k$) (Line 6). We then update the dataflow fact accordingly via a transfer function (Line 7). Next we check if the updated dataflow fact $\mathcal{OUT}'_k$ is different from that (*i.e.*, $\mathcal{OUT}_k$) at previous superstep. If so, the propagation is employed to update the dataflow fact as the newly computed value (Line 9). At the same time, $\mathcal{OUT}'_k$ is sent as a message to each of its successors d (Line 11), while activating d for the next superstep (Line 12).

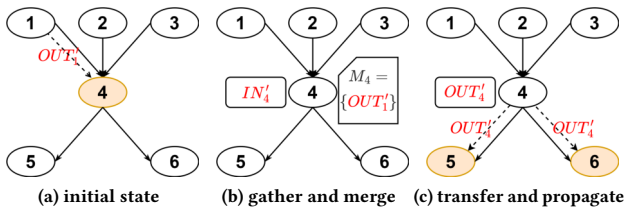


Figure 2: One superstep computation at vertex 4 in Algo. 4.

Example. Figure 2 illustrates the computation procedure at vertex 4 in Algorithm 4 for the same example as Figure 1. Suppose that at the beginning of a certain superstep, the active vertex 4's predecessor 1 has the newly updated fact $\mathcal{OUT}'_1$ and sends it as a message to 4 at the last superstep denoted by the dashed arrows in Figure 2a. The message is gathered as $\mathcal{M}_4 = \{\mathcal{OUT}'_1\}$ and then $\mathcal{IN}'_4$ is generated by incrementally merging the dataflow facts in $\mathcal{M}_4$ with $\mathcal{IN}_4$ shown as Figure 2b. Finally as shown by Figure 2c, $\mathcal{OUT}'_4$ is computed by performing transfer function on $\mathcal{IN}'_4$. Assuming that $\mathcal{OUT}'_4$ is different from $\mathcal{OUT}_4$, propagation is employed to send the newly updated $\mathcal{OUT}'_4$ to all the successors 5 and 6 while marking them as active.

3.3 Correctness Proof of Optimized Algorithm

The underlying rationale of such optimization is that the *merge* operation for the general monotone dataflow analysis satisfies the accumulative property. In other words, on each active vertex k, merging only the updated dataflow facts of k's predecessors with the old $\mathcal{IN}_k$ of last superstep should produce identical results to that merging the full set of dataflow facts of all its predecessors. The following Theorem 1 gives its formal definition.

Theorem 1 (**Accumulative Property**). Given an active vertex k at superstep i. Let $preds(k)$ be the set of k's predecessors. Without loss of generality, suppose at superstep $i-1$, a partial set of k's predecessors *i.e.*, $P'(k) \subseteq preds(k)$ update their outgoing dataflow facts, while the outgoing facts of the remaining *i.e.*, $P(k) = preds(k) - P'(k)$ stay unchanged. $\mathcal{IN}_k$ and $\mathcal{IN}'_k$ indicate the incoming dataflow fact of k at superstep $i-1$ and i, respectively. The accumulative property is satisfied if and only if the following equation holds.

$$\mathcal{IN}'_k \equiv \mathcal{IN}_k \otimes (\otimes_{p\in P'(k)} \mathcal{OUT}'_p)$$

i.e.,

$$(\otimes_{p\in P(k)} \mathcal{OUT}_p) \otimes (\otimes_{p\in P'(k)} \mathcal{OUT}'_p) \equiv (\otimes_{p\in preds(k)} \mathcal{OUT}_p) \otimes (\otimes_{p\in P'(k)} \mathcal{OUT}'_p)$$

Proof. Generally, there are two cases of monotone dataflow analysis, namely (1) increasing analysis with the join operator $\sqcup$ and (2) decreasing analysis with the meet operator $\sqcap$.

For case (1): $\otimes = \sqcup$ and for each predecessor $p \in P'(k)$, $\mathcal{OUT}_p \leq \mathcal{OUT}'_p$ holds where $\leq$ denotes the partial order relation and $\leq$ is reflexive, anti-symmetric and transitive according to its definition.

As defined by the $\sqcup$ operator which computes the least upper bound of two elements in the lattice, the following inequality 3.1 holds.

$$\mathcal{OUT}'_p \leq \mathcal{OUT}_p \sqcup \mathcal{OUT}'_p \tag{3.1}$$

Given that $\mathcal{OUT}_p \leq \mathcal{OUT}'_p$ (for increasing analysis) and $\mathcal{OUT}'_p \leq \mathcal{OUT}'_p$ ($\leq$ is reflexive), the following can be deduced:

$$\mathcal{OUT}_p \sqcup \mathcal{OUT}'_p \leq \mathcal{OUT}'_p \tag{3.2}$$

As $\leq$ is anti-symmetric, given inequalities 3.1 and 3.2 hold, we can imply the following equation 3.3.

$$\mathcal{OUT}'_p \equiv \mathcal{OUT}_p \sqcup \mathcal{OUT}'_p \tag{3.3}$$

Therefore, for all $p \in P'(k)$, the following equation 3.4 holds.

$$\sqcup_{p\in P'(k)} \mathcal{OUT}'_p \equiv \sqcup_{p\in P'(k)} (\mathcal{OUT}_p \sqcup \mathcal{OUT}'_p) \tag{3.4}$$

Because the $\sqcup$ operator in monotone dataflow analysis is both associative and commutative, we can imply that:

$$\sqcup_{p\in P'(k)} \mathcal{OUT}'_p \equiv (\sqcup_{p\in P'(k)} \mathcal{OUT}_p) \sqcup (\sqcup_{p\in P'(k)} \mathcal{OUT}'_p) \tag{3.5}$$

By joining $\sqcup_{p\in P(k)} \mathcal{OUT}_p$ with both sides of the equation 3.5, we can get the following:

$$(\sqcup_{p\in P(k)} \mathcal{OUT}_p) \sqcup (\sqcup_{p\in P'(k)} \mathcal{OUT}'_p) \equiv (\sqcup_{p\in P(k)} \mathcal{OUT}_p) \sqcup ((\sqcup_{p\in P'(k)} \mathcal{OUT}_p) \sqcup (\sqcup_{p\in P'(k)} \mathcal{OUT}'_p)) \tag{3.6}$$

And further equation 3.7 is deduced since $\sqcup$ is associative.

$$(\sqcup_{p\in P(k)} \mathcal{OUT}_p) \sqcup (\sqcup_{p\in P'(k)} \mathcal{OUT}'_p) \equiv ((\sqcup_{p\in P(k)} \mathcal{OUT}_p) \sqcup (\sqcup_{p\in P'(k)} \mathcal{OUT}_p)) \sqcup (\sqcup_{p\in P'(k)} \mathcal{OUT}'_p) \tag{3.7}$$

Since equation 3.8 holds,

$$(\sqcup_{p\in P(k)} \mathcal{OUT}_p) \sqcup (\sqcup_{p\in P'(k)} \mathcal{OUT}_p) \equiv \sqcup_{p\in preds(k)} \mathcal{OUT}_p \tag{3.8}$$

The final equation 3.9 for case (1) is thus proved.

$$(\sqcup_{p\in P(k)} \mathcal{OUT}_p) \sqcup (\sqcup_{p\in P'(k)} \mathcal{OUT}'_p) \equiv (\sqcup_{p\in preds(k)} \mathcal{OUT}_p) \sqcup (\sqcup_{p\in P'(k)} \mathcal{OUT}'_p) \tag{3.9}$$

Example. We use the vertex 4 in Figure 2 as an example to demonstrate the proof procedure. Assuming that $k = 4$, $preds(4) = \{1, 2, 3\}$. Given that at previous superstep, predecessor 1 updates its outgoing dataflow fact, thus $P'(4) = \{1\}$ and $P(4) = preds(4) - P'(4) = \{2, 3\}$. The incoming dataflow fact of 4 at previous and current supersteps are $\mathcal{IN}_4$ and $\mathcal{IN}'_4$, respectively. For case (1), suppose each dataflow fact corresponds to a set. The join operator $\sqcup$ indicates the set union $\cup$. The partial order relation $\leq$ is set inclusion $\subseteq$. Validating the accumulative property specific to this example is to prove the following equation holds:

$$\mathcal{IN}'_4 \equiv \mathcal{IN}_4 \cup \mathcal{OUT}'_1$$

Given the join operator $\cup$ and partial order relation $\subseteq$, it is apparent that the equation $\mathcal{OUT}'_1 \equiv \mathcal{OUT}_1 \cup \mathcal{OUT}'_1$ holds according to 3.1 and 3.2.

$$\begin{aligned} &\mathcal{OUT}'_1 \equiv \mathcal{OUT}_1 \cup \mathcal{OUT}'_1 \\ \overset{3.6}{\Longrightarrow}\ &(\mathcal{OUT}_2 \cup \mathcal{OUT}_3) \cup \mathcal{OUT}'_1 \equiv (\mathcal{OUT}_2 \cup \mathcal{OUT}_3) \cup (\mathcal{OUT}_1 \cup \mathcal{OUT}'_1) \\ \overset{3.7}{\Longrightarrow}\ &(\mathcal{OUT}_2 \cup \mathcal{OUT}_3) \cup \mathcal{OUT}'_1 \equiv (\mathcal{OUT}_2 \cup \mathcal{OUT}_3 \cup \mathcal{OUT}_1) \cup \mathcal{OUT}'_1 \\ \overset{3.8}{\Longrightarrow}\ &\mathcal{IN}'_4 \equiv \mathcal{IN}_4 \cup \mathcal{OUT}'_1 \end{aligned}$$

For case (2): $\otimes = \sqcap$ and for each predecessor $p \in P'(k)$, $\mathcal{OUT}'_p \leq \mathcal{OUT}_p$ holds. We can follow the similar proof logic.

As the meet $\sqcap$ operator calculates the greatest lower bound of elements, the following inequality 3.10 holds.

$$\mathcal{OUT}_p \sqcap \mathcal{OUT}'_p \leq \mathcal{OUT}'_p \tag{3.10}$$

Given that $\mathcal{OUT}'_p \leq \mathcal{OUT}_p$ (for decreasing analysis) and $\mathcal{OUT}'_p \leq \mathcal{OUT}'_p$ ($\leq$ is reflexive), the following can be deduced:

$$\mathcal{OUT}'_p \leq \mathcal{OUT}_p \sqcap \mathcal{OUT}'_p \tag{3.11}$$

As $\leq$ is anti-symmetric, given inequalities 3.10 and 3.11, the following equation 3.12 can be concluded.

$$\mathcal{OUT}'_p \equiv \mathcal{OUT}_p \sqcap \mathcal{OUT}'_p \tag{3.12}$$

Therefore, we can imply the following equations.

$$\begin{aligned}
&\sqcap_{p\in\mathcal{P}'(k)} \mathcal{OUT}'_p \equiv \sqcap_{p\in\mathcal{P}'(k)}(\mathcal{OUT}_p \sqcap \mathcal{OUT}'_p)\\
\Rightarrow\ &\sqcap_{p\in\mathcal{P}'(k)} \mathcal{OUT}'_p \equiv (\sqcap_{p\in\mathcal{P}'(k)} \mathcal{OUT}_p) \sqcap (\sqcap_{p\in\mathcal{P}'(k)} \mathcal{OUT}'_p)\\
\Rightarrow\ &(\sqcap_{p\in P(k)} \mathcal{OUT}_p) \sqcap (\sqcap_{p\in\mathcal{P}'(k)} \mathcal{OUT}'_p) \equiv\\
&(\sqcap_{p\in P(k)} \mathcal{OUT}_p) \sqcap ((\sqcap_{p\in\mathcal{P}'(k)} \mathcal{OUT}_p) \sqcap (\sqcap_{p\in\mathcal{P}'(k)} \mathcal{OUT}'_p))\\
\Rightarrow\ &(\sqcap_{p\in P(k)} \mathcal{OUT}_p) \sqcap (\sqcap_{p\in\mathcal{P}'(k)} \mathcal{OUT}'_p) \equiv\\
&((\sqcap_{p\in P(k)} \mathcal{OUT}_p) \sqcap (\sqcap_{p\in\mathcal{P}'(k)} \mathcal{OUT}_p)) \sqcap (\sqcap_{p\in\mathcal{P}'(k)} \mathcal{OUT}'_p)\\
\Rightarrow\ &(\sqcap_{p\in P(k)} \mathcal{OUT}_p) \sqcap (\sqcap_{p\in\mathcal{P}'(k)} \mathcal{OUT}'_p) \equiv\\
&(\sqcap_{p\in preds(k)} \mathcal{OUT}_p) \sqcap (\sqcap_{p\in\mathcal{P}'(k)} \mathcal{OUT}'_p) \qquad (3.13)
\end{aligned}$$

As equations 3.9 and 3.13 hold for each case, we ultimately complete the proof of Theorem 1.

□

4 IMPLEMENTATION

We implemented BigDataflow by following the distributed worklist algorithm on top of Apache Giraph 1.4.0[1], a well-maintained open source Java implementation of Pregel [6, 39].

Giraph replicates Pregel's concepts and adds several new features to this model, including master computation, out-of-core computation, and sharded aggregators, etc. In particular, Giraph first divides the input graph into a number of partitions based on Hadoop distributed file system. Within each superstep of the BSP model, Giraph launches multiple workers and enables each worker to process a partition separately in a distributed way. Giraph offers multiple effective partitioning schemes, which BigDataflow directly adopts to achieve good workload balance and scalability.

Besides, BigDataflow leverages two extra options offered by Giraph to realize the pulled-based worklist algorithm. (1) BasicComputation Class. BasicComputation is a general option for performing computations in Giraph. It can be used to access the graph's information, such as the superstep ID and information of vertices and edges. We extend it to distinguish analysis phase and acquire edge information in the implementation of BigDataflow. (2) Broadcast Class. Broadcast is the simplest way for master node to communicate with worker nodes in the scope of the entire cluster, ensuring that all vertices access the same information. BigDataflow exploits this feature to broadcast workers of entry nodes in CFG.

5 PROGRAMMING MODEL

BigDataflow as a framework supporting the general interprocedural dataflow analysis, provides a set of necessary APIs to users. Users readily implement a particular client analysis based on these APIs by specifying the information of input CFG, the dataflow equations (*i.e.*, merge, transfer), and the propagation logic. In the following, we first discuss the crucial APIs provided by BigDataflow, then demonstrate how to implement a client analysis based on the APIs.

5.1 APIs

Given a control flow graph or other sparse representation[16, 33], BigDataflow takes it as input and constructs the graph in memory. During a dataflow analysis, each vertex in the CFG maintains a dataflow fact, as well as the program statements associated. Lines 1-4 in listing 1 show the abstract class of `VertexAttribute`, which defines two members: dataflow fact of abstract class `Fact` and statements of class `Stmts`. Dataflow fact describes the dataflow information computed at each program point during analysis. The abstract class `Fact` (Line 6) leaves users the interface for specifying a particular type of dataflow fact in a client analysis. `Stmts` (Lines 9-11) describes the set of statements associated with the vertex, which determines the logic of transfer functions. In a statement-level dataflow analysis, dataflow fact is associated with each statement, where an instance of `Stmts` contains one single statement. While in a basic block-level analysis, each instance of `Stmts` indicates a set of statements in a basic block.

Listing 1: The APIs.

```
1  abstract class VertexAttribute{
2     Fact fact;
3     Stmts stmts;
4  }
5
6  abstract class Fact{}
7
8  abstract class Stmt {}
9  class Stmts {
10    Stmt[] stmts;
11 }
12
13 interface Analysis {
14    Fact merge(Set<Fact> predFacts, Fact oldIN);
15    Fact transfer(Stmts stmts, Fact inFact);
16    boolean propagate(Fact oldFact, Fact newFact);
17 }
```

Besides the above crucial data structures, three necessary components of dataflow analysis are defined in the `Analysis` interface shown as Lines 13-17 in listing 1. Whenever the computation on a vertex k is launched, `merge()` is first invoked to take the newly updated dataflow facts of predecessors together with the old incoming fact, and produce a new incoming dataflow fact for k. In general, the merge operation can be union or intersection depending on the specific client analysis. Users override `merge()` to specify the exact logic. Taking the incoming dataflow fact produced by `merge()` and the statements as input, `transfer()` computes the outgoing dataflow fact accordingly. Users are required to specify the particular transformation logic by overriding `transfer()` for a particular client analysis. `propagate()` describes the conditions for propagating dataflow facts to successors. Usually, propagation is decided by the comparison between old fact and new fact. User overrides `propagate()` to define concrete termination condition.

5.2 An Example of Alias Analysis

We use a context- and flow-sensitive alias analysis as an example to illustrate how to use the APIs to implement a client analysis. Flow-sensitive alias analysis computes the alias relations between pointer variables at each program point. As a fundamental analysis, it has been widely used in various applications including bug detection, security enforcement, optimizations, etc.

We adopt function cloning to achieve context-sensitivity [9, 53]. The input CFG to BigDataflow actually corresponds to a cloned interprocedural CFG. Taking the inlined ICFG as input, we first define a particular subclass `AliasStmt` to instantiate each statement for alias analysis. Its detailed implementation is omitted due to

[1] https://giraph.apache.org/

space limit. `Stmts` has only one `Stmt` instance as we would like to analyze the alias information at the granularity of statement. Here we adopt the program expression graph (PEG) [50] as a dataflow fact to represent the alias information at each program point. As such, each object of `Fact` is instantiated as a `PEG` instance. Next, `merge()` is achieved as union of the updated PEGs from predecessors with the old incoming fact. Within the overridden `transfer()`, edge addition and/or deletion are performed on PEG according to the semantics of each type of statement. If the old PEG and newly updated PEG are isomorphic, `propagate()` returns false and the vertex becomes inactive.

Listing 2: The implementation of flow-sensitive alias analysis on top of BigDataflow.

```
public class AliasStmt extends Stmt{...}
class AliasVertexAttribute extends VertexAttribute
{
   super();
   fact  = new PEG();
}

class AliasAnalysis implements Analysis{
   Fact merge(Set<Fact> predFacts, Fact oldIN) {
      PEG peg = (PEG)oldIn;
      for (Fact item : predFacts) {
         if (item == null)  continue;
         PEG prePEG = (PEG)item;
         peg.merge(prePEG);
      }
      return peg;
   }
   Fact transfer(Stmts stmts, Fact fact) {
      PEG peg = (PEG)fact;
      switch (stmts[0].getType()) {
         case Load:
           transfer_load(peg,(AliasStmt)stmts[0]);
           break;
         //...
      }
      return peg;
   }
   boolean propagate(Fact oldFact, Fact newFact) {
      if(oldFact == null)  return true;
      PEG newPEG = (PEG)newFact;
      PEG oldPEG = (PEG)oldFact;
      return !newPEG.consistent(oldPEG);
   }
}
```

As can be seen, to implement a client analysis on top of BigDataflow, users only need to specify the necessary functionalities specific to client analysis, without worrying about any implementation details of the underlying worklist algorithm as well as other system-side optimizations.

6 EVALUATION

Our evaluation focuses on the following three questions:

- Q1: What is the overall performance of BigDataflow given a rich set of distributed computing resources? (§ 6.1)
- Q2: How does BigDataflow perform compared with other competitive analysis systems/tools? (§ 6.2)
- Q3: What about the performance of BigDataflow given the varying numbers of cores and resources? (§ 6.3)

Subjects. To measure the performance of BigDataflow on scaling large programs, we selected five real-world software as the experimental subjects, including Linux kernel, Firefox, PostgreSQL, OpenSSL, and Apache Httpd. Table 1 lists detailed information about the subjects, such as the version (Version), the number of lines of code (#LoC), the number of functions (#Functions), and its description.

Table 1: Characteristics of subject programs.

Subject	Version	#LoC	#Functions	Description
Linux	5.2	17.5M	565K	operating system
Firefox	67.0	7.9M	770K	web browser
PostgreSQL	12.2	1.0M	30K	database system
OpenSSL	1.1.1	519K	12K	TLS protocol
Httpd	2.4.39	196K	6K	web server

Reference Tools. To validate the advantage of BigDataflow in terms of performance and scalability on large-scale programs, we selected the existing parallel/distributed analysis systems/tools as the competitors. For parallel algorithms, we chose Chianina [53], the most recent and state-of-the-art parallel system scaling context - and flow-sensitive analysis to large-scale C programs. Chianina is implemented in C/C++, and leverages two-level parallel computation model and out-of-core disk support to achieve both analysis efficiency and scalability. We ignore other sequential analysis algorithms [16, 45] since it has been validated that Chianina outperforms them [53]. For distributed work, since there exist no distributed systems supporting dataflow analysis, we used BigDataflow-classic, the version implemented based on the distributed classic worklist algorithm shown as Algorithm 3 as the reference tool. By default, BigDataflow is implemented using the optimized version (*i.e.*, Algorithm 4).

Hardware and Software Settings. All experiments were conducted in the Alibaba Cloud environment. Both BigDataflow and BigDataflow-classic are deployed on a cluster consisting of 125 Elastic Compute Service (ECS)[2] nodes with Alibaba Elastic MapReduce (EMR) installed. Each node (in particular *ecs.r7.2xlarge*) is equipped with 8 virtual CPU cores based on Intel Xeon Scalable processors and 64GB memory, running CentOS 7.4. The adopted EMR version is 3.14.0 corresponding to Hadoop 2.7.2 and Giraph 1.4.0. To compare with Chianina which can only run on a single-machine with shared memory, we used the most powerful server node available in the US (Virginia) region, *i.e.*, *ecs.r6.26xlarge* with 104 virtual cores, 768G memory, and 1T SSD-backed cloud disk.

Client Analyses. In the experiments, we implemented two client analyses, namely context-sensitive flow-sensitive alias analysis and instruction cache analysis, on top of BigDataflow, BigDataflow-classic, and Chianina. The alias analysis is same as the example discussed in § 5.2. For cache analysis, we followed the abstract model of LRU caches in [26] that adopts the set-associative organization. The configuration is set as 512 cache lines with LRU replacement strategy enabled. The analysis computes a cache model at each program point and decides a cache hit or miss. We chose the above two analyses for several reasons: 1) both analyses are fundamental and widely-used; 2) they are expensive and hardly scalable given their memory-intensive dataflow fact and compute-intensive transfer function; 3) they fall into the two cases of the

[2] https://www.alibabacloud.com/product/ecs

Table 2: Overall performance: columns #PAliases and #BCached indicate the number of alias pairs and the number of potentially cached memory blocks; columns #Workers, #PMem, Time and Cost represent the number of workers used, the size of peak memory consumed, the total analysis time, and the rental cost of cloud resources, respectively; #Part. indicates the number of partitions; - indicates out-of-memory error; (a) and (b) report the results for alias and cache analysis, respectively.

(a) Alias Analysis

		BigDataflow				BigDataflow-classic				Chianina			
Subject	#PAliases	#Workers	#PMem	Time	Cost	#Workers	#PMem	Time	Cost	#Part.	#PMem	Time	Cost
Linux	12.5B	350	3.5T	16.7mins	$15.8	350	-	-	-	4	453.4G	17.4hrs	$110.4
Firefox	11.5B	140	1.2T	16.5mins	$15.6	140	-	-	-	4	131.6G	5.3hrs	$33.6
PostgreSQL	727.0M	50	329.7G	2.8mins	$2.6	50	330.7G	4.9mins	$4.6	1	61.9G	50.4mins	$5.3
OpenSSL	734.8M	30	285.3G	3.5mins	$3.3	30	329.8G	6.8mins	$6.4	1	43.2G	35.4mins	$3.7
Httpd	183.1M	10	119.9G	2.8mins	$2.6	10	137.9G	4.0mins	$3.8	1	14.2G	11.2mins	$1.2

(b) Instruction Cache Analysis

		BigDataflow				BigDataflow-classic				Chianina			
Subject	#BCached	#Workers	#PMem	Time	Cost	#Workers	#PMem	Time	Cost	#Part.	#PMem	Time	Cost
Linux	21.5B	500	5.6T	44.4mins	$42.0	500	-	-	-	4	555.4G	9.4hrs	$59.6
Firefox	15.8B	400	4.4T	39.0mins	$36.9	400	-	-	-	4	351.5G	7.2hrs	$45.7
PostgreSQL	1.4B	180	1.1T	3.2mins	$3.0	180	1.1T	6.5mins	$6.1	1	115.3G	38.1mins	$4.0
OpenSSL	2.8B	180	1.3T	6.9mins	$6.5	180	1.5T	13.4mins	$12.7	1	227.6G	1.7hrs	$10.8
Httpd	782.0M	100	684.3G	3.0mins	$2.8	100	781.3G	4.6mins	$4.4	1	58.3G	18.5mins	$2.0

accumulative property in § 3.3 respectively, thereby validating the proof more comprehensively.

The context-sensitivity is achieved via fully function cloning (*i.e.*, ∞-CFA). We start the cloning based upon a call graph constructed by using a lightweight inclusion-based context-insensitive pointer analysis with support for function pointers. To handle recursion, we first identify the strongly connected components (SCCs) over the pre-computed call graph. Functions not in any SCC enjoy full context sensitivity. Whereas, level-2 call-string sensitivity (*i.e.*, using 2 top-most callsites as the distinguishing context) is used for those within SCCs. Note that function cloning is NOT the core contribution of this work. Users can adopt the classical k-limited context-sensitivity or other selective context-sensitivity techniques [17, 24]. This can be done by launching a cheap pre-analysis to understand the contexts desired, and then performing selective function cloning.

For each client analysis, the version implemented on top of BigDataflow, BigDataflow-classic and Chianina are identical and possess the same analysis precision. We checked the analysis results of three tools and validated they are consistent. Specifically, we compared the total number of alias pairs (including both memory alias and value alias) generated for alias analysis, and the total number of potentially cached memory blocks for cache analysis. The columns #PAliases and #BCached in Table 2 list the exact numbers.

6.1 Overall Performance

Tables 2a and 2b demonstrate the performance of BigDataflow when analyzing the five real-world subjects. Columns #Workers, #PMem, and Time indicate the number of workers used (one worker corresponding to one physical core), the amount of peak memory consumed, and the total analysis time, respectively.

It is well known that the complexity of a particular dataflow analysis is heavily dependent on many factors, such as the size, density, structure of the control flow graph, and the semantics of program under analysis. Thereby, it is difficult to give a general formula that can figure out the ideal number of workers needed. What we can do is to estimate a number as small as possible so as to the analysis task can be completed successfully and efficiently. To this end, we first run a small sample of the analysis (e.g., 1/50 of the input graph) on a small test cluster with 10 nodes. Based on the resource utilization data monitored, we estimate an initial number roughly. Next, we run the analysis on the initial number of workers. If the task fails due to insufficient memory, the number of workers is doubled until the analysis can succeed.

As can be seen, the peak memory consumed in both alias analysis and instruction cache analysis can easily reach several terabytes for large-scale programs, such as the Linux kernel and Firefox, due to the memory-intensive dataflow fact and the huge number of program points. Even for the smallest subject Httpd, performing the context- and flow-sensitive analysis takes more than a hundred or even several hundreds of gigabytes. This is consistent with the claim in [1] that memory would be the major bottleneck for analysis to scale to large programs. By leveraging the enormous amount of memory and computing resources in a cloud environment, BigDataflow manages to analyze all the subjects successfully and efficiently. The alias analysis can be completed within 20 minutes for all subjects; the more expensive cache analysis takes less than 45 minutes for the Linux kernel with 500 workers.

6.2 Comparison with Other Frameworks

Given the identical version of the client analysis implemented, we compared BigDataflow against BigDataflow-classic and Chianina with respect to performance and cost. Columns under BigDataflow-classic and Chianina in Table 2a and 2b show the detailed results of BigDataflow-classic and Chianina, respectively.

Chianina. As Chianina can only run on a single-machine with shared memory, we rented the most powerful server node with 104 virtual cores, 768G memory, and 1T SSD available in the US (Virginia) region of Alibaba Cloud. In terms of analysis time, Chianina

with 104 threads takes more than 17 hours and 9 hours to finish alias analysis and cache analysis over Linux. While BigDataflow completes alias and cache analysis within 20 and 45 minutes under a cluster, respectively. It shows that distributed parallelism enabled by BigDataflow indeed accelerates the analysis significantly (up to 62x and 12x for alias analysis and cache analysis on Linux, respectively). Note that BigDataflow takes more time for cache analysis than alias analysis on all the subjects, whereas Chianina does not. This can be explained from two aspects. First, cache analysis is more memory-intensive than alias analysis. The cache analysis on BigDataflow implemented in Java deservedly pays more GC time. Second, as observed, the alias analysis running on Chianina has low CPU utility due to load imbalance and excessive thread-switching costs for certain subjects (*e.g.*, Linux) when a large number of threads are enabled on a single machine.

As the computing resources used by BigDataflow and Chianina are different, we cannot simply derive that BigDataflow outperforms Chianina. For the sake of fairness, we measured the exact amount of rental costs of cloud resources in dollars paid by BigDataflow and Chianina for completing the identical analysis. As cloud providers generally adopt a unified pricing strategy, there is little difference in the price of nodes with similar resources across different providers. Without loss of generality, we calculated the cost by multiplying the analysis time and the official pay-as-you-go hourly price of Alibaba Cloud in US (Virginia) region[3]. In particular, at the time of submission, each node *ecs.r7.2xlarge* used by BigDataflow takes $0.454/hour. The price of the entire cluster is 0.454*125, *i.e.*, $56.75/hour. The single *ecs.r6.26xlarge* server node used by Chianina takes $6.344/hour. The cost columns in Table 2 show the detailed results. As can be seen, BigDataflow spends lower rental costs than Chianina over all the subjects except for Httpd. Although the price of the cluster used by BigDataflow ($56.75/hour) is much higher than that of the single server used by Chianina ($6.344/hour), BigDataflow takes much less time to finish the analysis than Chianina. We can thus conclude that BigDataflow is able to offer significantly higher analysis efficiency for large-scale programs, while taking fewer costs compared to Chianina.

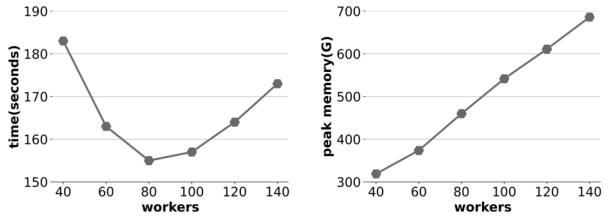
Regarding memory consumption, BigDataflow apparently consumes much more memory than Chianina. There are several reasons. (1) Chianina is a disk-based system where the memory consumption is strongly restricted. It will leverage disks to maintain the huge amount of data once the memory consumption exceeds a certain threshold. In contrast, BigDataflow prefers utilizing the memory on each node to perform communications and accelerate the analysis. (2) BigDataflow is implemented in Java, while Chianina is implemented in C/C++. No doubt Chianina would have less memory footprint than BigDataflow. (3) BigDataflow is running on top of Giraph. To achieve fault tolerance, Giraph needs to maintain extra (*e.g.*, 3) replicas for all the data stored. Moreover, for certain global data used in the analysis, BigDataflow has to broadcast it on every node, leading to extra memory consumption.

BigDataflow-classic. As numerous redundant and expensive dataflow facts were transmitted in the network and gathered at each vertex, BigDataflow-classic failed to analyze the large-scale subjects in our experiments (*i.e.*, Linux and Firefox) given the same computing resources as BigDataflow. It validates that BigDataflow does save memory resources, thus offering better scalability than BigDataflow-classic. For the analyses which both BigDataflow-classic and BigDataflow successfully complete, BigDataflow exclusively outperforms BigDataflow-classic in terms of time efficiency. This is because BigDataflow-classic requires more data transferred and merged than BigDataflow to accomplish the same analysis.

[3]https://www.alibabacloud.com/zh/product/ecs-pricing-list/en

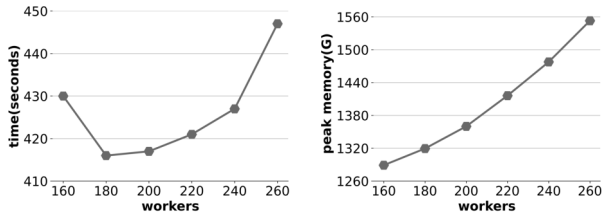
6.3 Scalability

To understand the scalability of BigDataflow, we measured the analysis time in seconds and peak memory consumption in gigabytes for both alias analysis and cache analysis given different numbers of workers. Figures 3 and 4 show the detailed performance results of alias analysis and cache analysis on OpenSSL, respectively, where the x-axis indicates the number of workers, and y-axis represents the time or peak memory used. Here only the data of PostgreSQL is reported. Other subjects show a similar trend to that of OpenSSL.



(a) time (b) peak memory

Figure 3: The time (a) and peak memory (b) used for alias analysis on OpenSSL with varying number of workers.



(a) time (b) peak memory

Figure 4: The time (a) and peak memory (b) used for cache analysis on OpenSSL with varying number of workers.

For alias analysis, the time taken by BigDataflow follows a V-bottom pattern shown as Figure 3a. When less workers are available (*i.e.*, 40), the total memory capacity just satisfies the analysis need. With the number of workers increasing from 40 to 80, increased parallelism is translated to higher performance. Therefore, the overall running time shows a descending trend. However, the communication cost among workers is monotonically increased with the growth of workers involved. Once the performance benefit of parallelism is no longer superior to the increased communication cost among workers, time climbs steadily. As such, for the specific analysis, having 80 workers provides the best trade-off between parallelism benefit and communication cost, leading to the shortest running time of all the tested parallel schedules. It implies that in practice we can seek a sweet spot of parallelism for different

subjects according to the tendency of running time as the number of workers changes. This is particularly meaningful because 1) cloud resources are on demand and charged on actual usage; and 2) performing dataflow analysis on the same large-scale program could be an iterative process as the program evolves constantly. In terms of the peak memory usage, as more threads/processes consume more memory space, it is not surprising that it shows an ascending trend with the growth of workers in Figure 3b. Figure 4 shows similar trends for cache analysis. As can be read from Figure 4a, with 160 workers available, BigDataflow successfully finishes the cache analysis. The best performance is achieved given more workers (*i.e.*, 180). After that point, more analysis time is needed with the increasing number of workers.

7 DISCUSSION

Usage Scenarios. BigDataflow offers the distributed capability in lifting sophisticated dataflow analysis to large-scale programs. It's highly valuable for organizations with large codebases to analyze, while often with their own cluster deployed. In such scenarios, BigDataflow readily offers both high-speed and scalable analysis to ultra-large-scale programs.

Soundness. Like other analysis frameworks (e.g., Soot and WALA), users implement a particular client analysis by specifying its corresponding merge and transfer functions. It is the analysis developer's responsibility to ensure the soundness of their analysis; BigDataflow faithfully executes whatever has been implemented by the developers. As for the underlying framework, we adopted the classic worklist algorithm [18], and directly implemented it based on the vertex-centric computation model. Its soundness stays unchanged.

8 RELATED WORK

8.1 Parallel and Distributed Static Analysis

Over the past decades, a few attempts have been made to speed up static program analysis by leveraging parallel and distributed computing facilities. Lee and Ryder [23] exploited algorithmic parallelism to accelerate dataflow analysis. Rodriguez et al.[36] proposed a parallel algorithm for IFDS-based dataflow analysis [35] based on the actor model, which requires the transfer function to be distributive over meet operators. Nagaraj and Govindarajan [31] utilized Intel Threading Building Blocks to design a parallel flow-sensitive pointer analysis algorithm. Su et al. [44] proposed parallel CFL-reachability-based flow-insensitive pointer analysis. Importantly, all the above approaches rely heavily on memory for computation. They can rarely scale to large systems such as Linux kernel.

Following the line of systemizing program analysis, various systems are developed to support scalable interprocedural analysis. Graspan [46, 52] and BigSpa [51] scale the context-sensitive CFL-reachability analysis [34] in a single machine and distributed environment, respectively. Unfortunately, many dataflow analyses cannot fall into this category, such as cache analysis and interval analysis. Chianina [53] is an out-of-core graph system performing the context- and flow-sensitive analyses in parallel. However, restricted by the limited parallel computing resources in a single machine, it is inefficient when analyzing large-scale systems code.

For distributed work, Albarghouthi et al. [2] took the inspirations from MapReduce paradigm and parallelized the demand-driven top-down analyses, such as verification and software model checking. They failed to support dataflow analysis. Garbervetsky et al. [10] recently devised a distributed worklist algorithm based on the actor model. However, it does not support the standard dataflow analysis due to the absence of flow ordering between actors. Christakis et al. [8] explored input splitting strategies to analyze different code pieces on parallel partitions independently. However, as stated explicitly, the splitting causes analysis imprecision due to the information loss across separate partitions. Greathouse et al. [13] extended dynamic dataflow analyses with a novel sampling system to achieve low runtime overhead. Apparently, they only focused on dynamic analysis rather than static dataflow analysis.

8.2 Vertex-Centric Graph Processing

Vertex-centric model has been tightly incorporated into distributed processing frameworks to tackle the challenge of large-scale graph processing. Based on that, Pregel [27] is the pioneering system supporting general graph applications. Pregel adopts BSP model to accelerate the intensive computation. Following the idea of Pregel, Apache Giraph [4] is implemented in Java as an open source system. Following Pregel, more advanced vertex-centric models and variants have been proposed. GraphLab [25] supports asynchronous vertex computation based on Chandy-Lamport snapshots without halting the entire program. GraphX [12] is a graph system based on Resilient Distributed Dataset (a.k.a., RDD) abstraction.

Note that all the above graph systems are dedicated to the general graph applications. None of them can directly scale the interprocedural dataflow analysis well. As a result, we propose BigDataflow, the first distributed system tailored to dataflow analysis.

9 CONCLUSION

This paper proposes a distributed interprocedural dataflow analysis framework named BigDataflow. By leveraging the large amount of memory and CPU cores in the cloud, BigDataflow greatly improves the scalability of dataflow analysis for analyzing large-scale programs. The experiments conducted in a real-world cloud environment validate that BigDataflow not only scales context-sensitive dataflow analysis to million lines of code, but also completes such analysis in a highly efficient manner. It can be expected that we could achieve nearly on-the-fly analysis of industrial-scale codebases by leveraging modern cloud computing facilities.

10 DATA AVAILABILITY

BigDataflow is publicly available: https://github.com/BigDataflow-system. All the experimental data can be accessed via the link: https://figshare.com/articles/dataset/material_fse23_zip/21971945/3.

ACKNOWLEDGMENTS

We would like to thank all the anonymous reviewers for their valuable comments. This work is partially supported by the National Natural Science Foundation of China (Grant No. 62272217), the Fundamental Research Funds for the Central Universities (Grant No. 020214380104) and Alibaba Group via the Alibaba Innovation Research (AIR) program.

REFERENCES

[1] Alex Aiken, Suhabe Bugrara, Isil Dillig, Thomas Dillig, Brian Hackett, and Peter Hawkins. [n. d.]. An Overview of the Saturn Project. In *PASTE* (San Diego, California, USA) *(PASTE '07)*. ACM, 43–48. https://doi.org/10.1145/1251535.1251543

[2] Aws Albarghouthi, Rahul Kumar, Aditya V. Nori, and Sriram K. Rajamani. 2012. Parallelizing Top-down Interprocedural Analyses. In *PLDI* (Beijing, China) *(PLDI '12)*. ACM, 217–228. https://doi.org/10.1145/2254064.2254091

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *PLDI* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, 259–269. https://doi.org/10.1145/2594291.2594299

[4] Ching Avery. 2011. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara* 11, 3 (2011), 5–9.

[5] Craig Chambers, Jeffrey Dean, and David Grove. 1996. *Frameworks for Intra- and Interprocedural Dataflow Analysis.* Technical Report. University of Washington.

[6] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One Trillion Edges: Graph Processing at Facebook-Scale. *Proc. VLDB Endow.* 8, 12 (aug 2015), 1804–1815. https://doi.org/10.14778/2824032.2824077

[7] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. 1991. Automatic Construction of Sparse Data Flow Evaluation Graphs. In *POPL* (Orlando, Florida, USA) *(POPL '91)*. ACM, New York, NY, USA, 55–66. https://doi.org/10.1145/99583.99594

[8] Maria Christakis, Thomas Cottenier, Antonio Filieri, Linghui Luo, Muhammad Numair Mansur, Lee Pike, Nicolás Rosner, Martin Schäf, Aritra Sengupta, and Willem Visser. 2022. Input Splitting for Cloud-Based Static Application Security Testing Platforms. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1367–1378. https://doi.org/10.1145/3540250.3558944

[9] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. 1994. Context-sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In *PLDI* (Orlando, Florida, USA) *(PLDI '94)*. ACM, New York, NY, USA, 242–256. https://doi.org/10.1145/178243.178264

[10] Diego Garbervetsky, Edgardo Zoppi, and Benjamin Livshits. 2017. Toward Full Elasticity in Distributed Static Analysis: The Case of Callgraph Analysis. In *ESEC/FSE* (Paderborn, Germany) *(ESEC/FSE 2017)*. ACM, 442–453. https://doi.org/10.1145/3106237.3106261

[11] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *OSDI* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, 17–30.

[12] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *OSDI* (Broomfield, CO) *(OSDI'14)*. USENIX Association, 599–613.

[13] Joseph L. Greathouse, Chelsea LeBlanc, Todd Austin, and Valeria Bertacco. 2011. Highly scalable distributed dataflow analysis. In *International Symposium on Code Generation and Optimization (CGO 2011)*. 277–288. https://doi.org/10.1109/CGO.2011.5764695

[14] Rong Gu, Zhiqiang Zuo, Xi Jiang, Han Yin, Zhaokang Wang, Linzhang Wang, Xuandong Li, and Yihua Huang. 2021. Towards Efficient Large-Scale Interprocedural Program Static Analysis on Distributed Data-Parallel Computation. *IEEE Trans. Parallel Distrib. Syst.* 32, 4 (apr 2021), 867–883. https://doi.org/10.1109/TPDS.2020.3036190

[15] Ben Hardekopf and Calvin Lin. 2009. Semi-Sparse Flow-Sensitive Pointer Analysis. In *POPL*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/1480881.1480911

[16] Ben Hardekopf and Calvin Lin. 2011. Flow-sensitive Pointer Analysis for Millions of Lines of Code. In *CGO (CGO '11)*. IEEE Computer Society, Washington, DC, USA, 289–298. http://dl.acm.org/citation.cfm?id=2190025.2190075

[17] Minseok Jeon, Sehun Jeong, and Hakjoo Oh. 2018. Precise and Scalable Points-to Analysis via Data-Driven Context Tunneling. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 140 (oct 2018), 29 pages. https://doi.org/10.1145/3276510

[18] John B. Kam and Jeffrey D. Ullman. 1976. Global Data Flow Analysis and Iterative Algorithms. *J. ACM* 23, 1 (Jan. 1976), 158–171. https://doi.org/10.1145/321921.321938

[19] John B. Kam and Jeffrey D. Ullman. 1977. Monotone Data Flow Analysis Frameworks. *Acta Inf.* 7, 3 (Sept. 1977), 305–317. https://doi.org/10.1007/BF00290339

[20] Arijit Khan. 2016. Vertex-Centric Graph Processing: The Good, the Bad, and the Ugly. (12 2016).

[21] Gary A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *POPL* (Boston, Massachusetts) *(POPL '73)*. ACM, New York, NY, USA, 194–206. https://doi.org/10.1145/512927.512945

[22] Sungho Lee, Julian Dolby, and Sukyoung Ryu. 2016. HybriDroid: Static analysis framework for Android hybrid applications. In *2016 31st (ASE)*. 250–261. https://doi.org/10.1145/2970276.2970368

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

[24] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Precision-Guided Context Sensitivity for Pointer Analysis. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 141 (oct 2018), 29 pages. https://doi.org/10.1145/3276511

[25] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. 2012. Distributed graphlab: A framework for machine learning in the cloud. *arXiv preprint arXiv:1204.6078* (2012).

[26] Mingsong Lv, Nan Guan, Jan Reineke, Reinhard Wilhelm, and Wang Yi. 2016. A Survey on Static Cache Analysis for Real-Time Systems. *Leibniz Trans. Embed. Syst.* 3, 1 (2016), 05:1–05:48. https://doi.org/10.4230/LITES-v003-i001-a005

[27] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-scale Graph Processing. In *SIGMOD* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. ACM, 135–146. https://doi.org/10.1145/1807167.1807184

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

[29] David Melski and Thomas W. Reps. 1997. Interconveritibility of Set Constraints and Context-Free Language Reachability. In *PEPM*. 74–89. https://doi.org/10.1145/258993.259006

[30] Brian R. Murphy and Monica S. Lam. 1999. Program Analysis with Partial Transfer Functions. In *PEPM* (Boston, Massachusetts, USA) *(PEPM '00)*. ACM, New York, NY, USA, 94–103. https://doi.org/10.1145/328690.328703

[31] Vaivaswatha Nagaraj and R Govindarajan. 2013. Parallel flow-sensitive pointer analysis by graph-rewriting. In *IEEE*. IEEE, 19–28. https://doi.org/10.1109/PACT.2013.6618800

[32] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 1999. *Principles of Program Analysis.* Springer-Verlag, Berlin, Heidelberg.

[33] G. Ramalingam. 2002. On Sparse Evaluation Representations. *Theor. Comput. Sci.* 277, 1-2 (April 2002), 119–147. https://doi.org/10.1016/S0304-3975(00)00315-7

[34] Thomas Reps. 1997. Program Analysis via Graph Reachability. In *ILPS* (Port Washington, New York, USA) *(ILPS '97)*. MIT Press, 5–19.

[35] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL* (San Francisco, California, USA) *(POPL '95)*. ACM, 49–61. https://doi.org/10.1145/199448.199462

[36] Jonathan Rodriguez and Ondřej Lhoták. 2011. Actor-based Parallel Dataflow Analysis. In *CC* (Saarbrücken, Germany) *(CC'11/ETAPS'11)*. 179–197.

[37] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from Building Static Analysis Tools at Google. *Commun. ACM* 61, 4 (March 2018), 58–66. https://doi.org/10.1145/3188720

[38] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, 598–608. https://doi.org/10.1109/ICSE.2015.76

[39] Roman Shaposhnik, Claudio Martella, and Dionysios Logothetis. 2015. *Practical Graph Analytics with Apache Giraph* (1st ed.). Apress, USA.

[40] M Sharir and A Pnueli. 1978. *Two approaches to interprocedural data flow analysis.* New York Univ. Comput. Sci. Dept., New York, NY. https://cds.cern.ch/record/120118

[41] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: Fast and Precise Sparse Value Flow Analysis for Million Lines of Code. In *PLDI* (Philadelphia, PA, USA) *(PLDI 2018)*. ACM, 693–706. https://doi.org/10.1145/3192366.3192418

[42] Olin Grigsby Shivers. 1991. *Control-flow Analysis of Higher-order Languages of Taming Lambda.* Ph. D. Dissertation. Pittsburgh, PA, USA. UMI Order No. GAX91-26964.

[43] Ting Su, Ke Wu, Weikai Miao, Geguang Pu, Jifeng He, Yuting Chen, and Zhendong Su. 2017. A Survey on Data-Flow Testing. *ACM Comput. Surv.* 50, 1, Article 5 (mar 2017), 35 pages. https://doi.org/10.1145/3020266

[44] Yu Su, Ding Ye, and Jingling Xue. 2014. Parallel pointer analysis with cfl-reachability. In *2014 43rd International Conference on Parallel Processing*. IEEE, 451–460. https://doi.org/10.1109/ICPP.2014.54

[45] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural Static Value-Flow Analysis in LLVM. In *CC* (Barcelona, Spain) *(CC 2016)*. Association for Computing Machinery, New York, NY, USA, 265–266. https://doi.org/10.1145/2892208.2892235

[46] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. [n. d.]. Graspan: A Single-machine Disk-based Graph System for Interprocedural Static Analyses of Large-scale Systems Code. In *ASPLOS* (Xi'an, China) *(ASPLOS '17)*. 389–404. https://doi.org/10.1145/3037697.3037744

[47] John Whaley and Monica S. Lam. 2004. Cloning-based Context-sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *PLDI* (Washington DC, USA) *(PLDI '04)*. ACM, New York, NY, USA, 131–144. https://doi.org/10.1145/996841.996859

[48] Robert P. Wilson and Monica S. Lam. 1995. Efficient Context-sensitive Pointer Analysis for C Programs. In *PLDI* (La Jolla, California, USA) *(PLDI '95)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/207110.207111

[49] Roland Wismüller. 1994. Debugging of Globally Optimized Programs Using Data Flow Analysis. In *PLDI* (Orlando, Florida, USA) *(PLDI '94)*. Association for Computing Machinery, New York, NY, USA, 278–289. https://doi.org/10.1145/178243.178430

[50] Xin Zheng and Radu Rugina. 2008. Demand-driven alias analysis for C. In *POPL*. 197–208. https://doi.org/10.1145/1328438.1328464

[51] Zhiqiang Zuo, Rong Gu, Xi Jiang, Zhaokang Wang, Yihua Huang, Linzhang Wang, and Xuandong Li. 2019. BigSpa: An Efficient Interprocedural Static Analysis Engine in the Cloud. In *IPDPS (IPDPS'19)*. https://doi.org/10.1109/IPDPS.2019.00086

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

[53] Zhiqiang Zuo, Yiyu Zhang, Qiuhong Pan, Shenming Lu, Yue Li, Linzhang Wang, Xuandong Li, and Guoqing Harry Xu. 2021. Chianina: an evolving graph system for flow-and context-sensitive analyses of million lines of C code. In *PLDI*. 914–929. https://doi.org/10.1145/3453483.3454085

Received 2023-02-02; accepted 2023-07-27