Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

Technical Report No. NJU-SEG-2021-IJ-002

2021-IJ-002

Systemizing Interprocedural Static Analysis of Large-scale Systems Code with Graspan

Zhiqiang Zuo, Kai Wang, Aftab Hussain, Ardalan Amiri Sani, Yiyu Zhang, Shenming Lu,

Wensheng Dou, Linzhang Wang, Xuandong Li, Chenxi Wang, Guoqing Harry Xu

Technical Report 2021

Systemizing Interprocedural Static Analysis of Large-scale Systems Code with Graspan

ZHIQIANG ZUO, State Key Laboratory for Novel Software Technology, Nanjing University, China
KAI WANG, University of California, Los Angeles, USA
AFTAB HUSSAIN and ARDALAN AMIRI SANI, University of California, Irvine, USA
YIYU ZHANG and SHENMING LU, State Key Laboratory for Novel Software Technology, Nanjing University, China
WENSHENG DOU, University of Chinese Academy of Sciences and State Key Lab of Computer Sciences, Institute of Software, Chinese Academy of Sciences, China
LINZHANG WANG and XUANDONG LI, State Key Laboratory for Novel Software Technology, Nanjing University, China
CHENXI WANG and GUOQING HARRY XU, University of California, Los Angeles, USA

There is more than a decade-long history of using static analysis to find bugs in systems such as Linux. Most of the existing static analyses developed for these systems are simple checkers that find bugs based on pattern matching. Despite the presence of many sophisticated interprocedural analyses, few of them have been employed to improve checkers for systems code due to their complex implementations and poor scalability.

In this article, we revisit the scalability problem of interprocedural static analysis from a "Big Data" perspective. That is, we turn sophisticated code analysis into *Big Data analytics* and leverage novel data processing techniques to solve this traditional programming language problem. We propose *Graspan*, a disk-based parallel graph system that uses an *edge-pair* centric computation model to compute *dynamic transitive closures* on very large program graphs. We develop two backends for Graspan, namely, *Graspan-C* running on CPUs and *Graspan-G* on GPUs, and present their designs in the article. Graspan-C can analyze large-scale systems code on any commodity PC, while, if GPUs are available, Graspan-G can be readily used to achieve orders of magnitude speedup by harnessing a GPU's massive parallelism.

We have implemented *fully context-sensitive* pointer/alias and dataflow analyses on Graspan. An evaluation of these analyses on large codebases written in multiple languages such as Linux and Apache Hadoop demonstrates that their Graspan implementations are language-independent, scale to millions of lines of

This work was partially supported by the National Natural Science Foundation of China (Nos. 61632015, 62032010, and 61802168), and the Natural Science Foundation of Jiangsu Province (No. BK20191247). Wensheng Dou was supported by the Frontier Science Project of Chinese Academy of Sciences (No. QYZDJ-SSW-JSC036). The UCLA authors were supported in part by the US National Science Foundation under grants CNS-1613023, CNS-1703598, CNS-1763172, CNS-2006437, CNS-2007737, and CNS-2106838, and the US Office of Naval Research under grants N00014-16-1-2913 and N00014-18-1-2037.
Authors' addresses: Z. Zuo (corresponding author), Y. Zhang, S. Lu, L. Wang, and X. Li, State Key Laboratory for Novel Software Technology, Nanjing University, China; emails: {zqzuo, lzwang, lxd}@nju.edu.cn; K. Wang, C. Wang, and G. H. Xu, University of California, Los Angeles; emails: {wangkai, wangchenxi, harryxu}@cs.ucla.edu; A. Hussain and A. A. Sani, University of California, Irvine; emails: {aftabh, ardalan}@ics.uci.edu; W. Dou, University of Chinese Academy of Sciences and State Key Lab of Computer Sciences, Institute of Software, Chinese Academy of Sciences, China; email: wensheng@iscas.ac.cn.

0734-2071/2021/07-ART4 $15.00
https://doi.org/10.1145/3466820

code, and are much simpler than their original implementations. Moreover, we show that these analyses can be used to uncover many real-world bugs in large-scale systems code.

CCS Concepts: • **Computer systems organization → Special purpose systems**; Reliability; • **Theory of computation → Program analysis**; • **Computing methodologies** → Massively parallel algorithms;

Additional Key Words and Phrases: static analysis, graph processing, disk-based systems

ACM Reference format:
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 (July 2021), 39 pages.
https://doi.org/10.1145/3466820

1 INTRODUCTION

Static analysis has been used to find bugs in systemssoftware for more than a decade now [10, 16, 17, 21, 24, 32, 36, 37, 39, 46, 85, 89, 103, 129, 139]. Based on a set of systems rules, a static checker builds patterns and inspects code statements to perform "pattern matching." If a code region matches one of the patterns, then a violation is found and reported. Static checkers have many advantages over recent, more advanced bug detectors based on SAT solvers or symbolic execution [21]: They are simple, easy to implement, and scalable. Furthermore, they produce deterministic and easy-to-understand bug reports compared to, for example, a symbolic execution technique, which often produces non-deterministic bug reports that are difficult to reason about [35].

1.1 Problems

Unfortunately, the existing static checkers use many heuristics when searching for patterns, resulting in missing bugs and/or reporting false warnings. For example, Chou et al. [30] and Palix et al. [89] developed nine checkers to find bugs in the Linux kernel. Most of these checkers generate both false negatives and false positives. For instance, their **Null** checker tries to identify NULL pointer dereference bugs by inspecting only the functions that directly return NULL. However, a NULL value can be generated from the middle of a function and propagated a long way before it is dereferenced at a statement. Such NULL value propagation will be missed entirely by the **Null** checker.

As another example, the **Inull/NullRef** checker [30, 89] checks whether a pointer dereference such as $a = b\text{->}\ f$ is *post-dominated* by a NULL test on the pointer such as `if`(b). The heuristic here is that if the developer checks whether b can be NULL after dereferencing b, then the dereferencing can potentially be on a NULL pointer. However, in many cases, the dereferencing occurs in one of the many control flow paths and in this particular path the pointer can never be NULL. The developer adds the NULL test simply because the NULL value may flow to the test point from a different control branch.

Our key observation in reducing the number of false positives and negatives reported by these checkers is to leverage *interprocedural analysis*. Among the aforementioned nine checkers, six that check flow properties can be easily improved (e.g., producing fewer false positives and false negatives) using an interprocedural analysis, as shown in Table 1.

While using interprocedural analyses to improve bug detection appears to be obvious, there seems to be a large gap between the state-of-the-art and the state-of-the-practice. On the one hand, the past decade has seen a large number of sophisticated and powerful analyses developed

Table 1. A Subset of Checkers Used by References [21, 30, 89] to Find Bugs in the Linux Kernel, their Target Problems, their Limitations, the Potential Ways to Improve Them Using a Sophisticated Interprocedural Analysis

Checker	Target Problems	Limitations	Potential Improvement with Interprocedural Analyses
Block	Deadlocks	Focus on "direct" invocations of the blocking functions (Negative)	Use a **pointer/alias analysis** to identify indirect invocations via function pointers of the blocking functions
Null	NULL pointer derefs	Inspect a closure of functions that return NULL explicitly (Negative)	Use a **dataflow analysis** to identify functions where NULL can be propagated through dataflows to their return variables
Range	Use user data as array index without checks	Only check indices directly from user data (Negative)	Use a **dataflow analysis** to identify indices coming transitively from user data as well
Lock/Intr	Double acquired locks and disabled interrupts not appropriately restored	Identify lock/interrupt objects by variable names (Negative)	Use a **pointer/alias analysis** to understand aliasing relationships among lock objects in different lock sites
Free	Use of a freed object	Identify freed/used objects by var names (Negative)	Use a **pointer/alias analysis** to check if there is aliasing between objects freed and used afterwards
Size	Inconsistent sizes between an allocated obj and the type of the RHS var	Only check alloc sites (Negative)	Use a **pointer/alias analysis** to identify other vars that *point to* the same object with an inconsistent type
Inull/NullRef	NULL pointer derefs	Report all derefs post-dominated by NULL tests (Positive)	Use a **dataflow analysis** to filter out cases where the involved pointers *must not* be NULL

Positive/negative indicates whether the limitation can result in false positives/negatives.

by program analysis researchers. On the other hand, none of these techniques are widely used to find bugs in systems software.

We believe that the reason is two-fold. First, an interprocedural analysis is often not scalable enough to analyze large codebases such as the Linux kernel. For such an analysis to be useful, it often needs to be *context-sensitive*, that is, distinct solutions need to be produced and maintained for different calling contexts (i.e., a chain of call sites representing a runtime call stack). However, the number of calling contexts grows *exponentially* with the size of the program and even a moderate-sized program can have as large as 10^{14} distinct contexts [132], making the analysis both compute- and memory-intensive. Furthermore, most interprocedural analyses are difficult to parallelize, because they frequently involve decision making based on information discovered *dynamically*. Thus, most of the existing implementations of such analyses are entirely sequential.

Second, the sheer implementation complexity scares practitioners away. Much of this complexity stems from optimizing the analysis rather than implementing the base algorithm. For example, in a widely used Java pointer analysis [116], more than three quarters of the code performs approximations to make sure some results can be returned before a user-given time budget runs out. The base algorithm implementation takes a much smaller portion. This level of tuning complexity simply does not align with the "simplest-working-solution" [64] philosophy of systems builders.

1.2 Insight

Our idea is inspired by the way a graph system enables scalable processing of large graphs. Graph system support pioneered by Pregel [76] provides a "one-stone-two-birds" solution, in which the optimization for scalability is mainly achieved by the (distributed or disk-based) system itself, requiring the developers to only write simple vertex programs using the interfaces provided by the system.

In this article, we demonstrate a similar "one-stone-two-birds" solution for interprocedural program analysis. Our key observation in this work is that many interprocedural analyses can be formulated as a *graph reachability* problem [95, 96, 105, 116, 142]. Pointer/alias analysis and dataflow analysis are two typical examples. In a pointer/alias analysis, if an object (e.g., created by a malloc) can directly or transitively reach a variable on a directed graph representation of the program, then the variable may point to the object. In a dataflow analysis that tracks NULL pointers, similarly, a transitive flow from a NULL value to a variable would make NULL propagate to the variable.

Therefore, we turn the programs into graphs and treat the analyses as graph traversal. This approach opens up opportunities to leverage parallel graph processing systems to analyze large programs efficiently.

This direction is particularly promising due to the following three benefits:

- **High Scalability:** Sophisticated analysis algorithms can be reduced to simple and mechanical data computations that can be automatically parallelized by the underlying system. By leveraging massive amounts of (CPU, GPU, and disk) resources, the system brings performance benefit, automatically, to a wide spectrum of program analysis workloads.
- **Easy Implementation:** The concern about efficiency and scalability is shifted from analysis developers' shoulders to the system. The implementation of a client analysis requires only the development of simple **user-defined functions (UDFs)**, enabling regular developers to easily prototype and maintain an analysis without worrying about how to tune its performance.
- **Multilingual Support:** The program under analysis is fed to the system as input data. With appropriate normalization, programs (or different components in the same program) written in different languages can be transformed into a graph with a unified format, allowing them to be analyzed by the same analysis implementation.

1.3 Existing Systems

Several graph systems are available today. These systems are either distributed (e.g., GraphLab [73], PowerGraph [43], or GraphX [44]) or single-machine-based (e.g., GraphChi [62], XStream [101], GridGraph [156], GraphReducer [106], or GTS [59]). Since program analysis is intended to assist developers to find bugs in their daily development tasks, their machines are the environments in which we would like our system to run, so developers can check their code on a regular basis without needing to access a cluster. Hence, disk-based systems naturally become our choice.

We initially planned to use an existing system to analyze program graphs. We soon realized that a ground-up redesign (i.e., from the programming model to the engine) is needed to build a system for analyzing large programs. The main reason is that the graph workload for interprocedural analyses is significantly different from a regular graph algorithm (such as PageRank) that iteratively performs computations on vertex values on a static graph. An interprocedural analysis, on the contrary, focuses on computing reachability by repeatedly adding *transitive edges*, rather than on updating vertex values. For instance, a pointer analysis needs to add an edge from each allocation vertex to each variable vertex that is transitively reachable from the allocation.

More specifically, many interprocedural analyses are essentially *dynamic reachability* problems in the sense that the addition of a new edge is guided by a constraint on the labels of the existing edges. In a static analysis, the label of an edge often represents the semantics of the edge (e.g., an assignment or a dereference). For two edges $a \xrightarrow{l_1} b$ and $b \xrightarrow{l_2} c$, a transitive edge from a to c is added only if the concatenation of l_1 and l_2 forms a string of a (context-free) grammar.

This constraint-guided reachability problem, in general, requires ***dynamic transitive closure* (DTC)** computation [51, 98, 141], which has a wide range of applications in program analysis and other domains. The DTC computation dictates two important abilities of the graph system. First, at each vertex, all of its incoming and outgoing edges need to be visible to perform label matching and edge addition. In the above example, when b is processed, both $a \xrightarrow{l_1} b$ and $b \xrightarrow{l_2} c$ need to be accessed to add the edge from a to c. This requirement immediately excludes edge-centric systems such as XStream [101] from our consideration, because these systems stream in edges in a random order and, thus, this pair of edges may not be simultaneously available.

Second, the system needs to support a large number of edges added dynamically. The added edges can be even more than the original edges in the graph. While vertex-centric systems such as GraphChi [62] support dynamic edge addition, this support is very limited. In the presence of a large number of added edges, it is critical that the system is able to (1) quickly check edge duplicates and (2) appropriately repartition the graph. Unfortunately, GraphChi supports neither of these features. Other graph systems including evolving and streaming graph systems [122, 123, 159] cannot support such features either.

1.4 Our Contributions

This article presents Graspan, the first *single machine, disk-based* parallel graph processing system tailored for interprocedural static analyses. Given a program graph and a grammar specification of an analysis, Graspan offers two major performance and scalability benefits: (1) the core computation of the analysis is automatically parallelized and (2) out-of-core support is exploited if the graph is too big to fit in memory. At the heart of Graspan is a parallel ***edge-pair* (EP)**-centric computation model that, in each iteration, loads two partitions of edges into memory and "joins" their edge lists to produce a new edge list. Whenever the size of a partition exceeds a threshold value, its edges are repartitioned. Graspan supports both in-memory (for small programs) and out-of-core (for large programs) computation. Joining of two edge lists is fully parallelized, allowing multiple transitive edges to be simultaneously added.

Graspan provides an intuitive programming model, in which the developer only needs to generate the graph and define the grammar that guides the edge addition, a task orders of magnitude easier than coming up with a well-tuned implementation of the analysis that would give trouble to skillful researchers for months.

To understand the performance potential of the proposed idea of "systemized" analyses, we implemented Graspan with two backends, one on CPU (Graspan-C) and a second on GPU (Graspan-G). Graspan-C analyzes large-scale systems code on a commodity PC, while Graspan-G utilizes GPU resources for significantly improved efficiency. The differences in the design and implementation of the two backends are transparent to users.

Recent work shows the effectiveness of backing static analyses with Datalog [20, 132] or Database [131]. While leveraging Datalog makes analysis implementations easier, the existing Datalog engines are designed in *generic* ways, i.e., not considering the characteristics of the program analysis workload. Furthermore, there does not exist any out-of-core Datalog engine that can process very large graphs on a single machine. For example, the Linux kernel program graph has more than 1B edges. The fastest shared memory Datalog engine SociaLite [63] quickly ran out of memory while Graspan-C processed it in several hours (cf. Section 6.5). While distributed Datalog engines such as Myria [125] and BigDatalog [111] are available, it is unrealistic to require developers to frequently access a cluster in their daily development.

We have implemented three client analyses,[1] namely, a fully context-sensitive field-insensitive pointer/alias analysis, a fully context-sensitive field-sensitive pointer/alias analysis, and a fully context-sensitive dataflow analysis on Graspan. Context-sensitivity is achieved by making aggressive inlining [107]. That is, we clone the body of a function for every single context leading to the function. This approach is feasible only because the out-of-core support in Graspan frees us from worrying about additional memory usage incurred by inlining. We treat the functions in recursions *context insensitively* by merging the functions in each strongly connected component on the call graph into one function without cloning function bodies.

[1]The analyses supported by Graspan are path-insensitive and flow-insensitive except for the IFDS-like [96] dataflow analysis.

Results. We have implemented Graspan in three languages: C/C++, Java, and CUDA; these implementations are all publicly available at https://github.com/Graspan. Graspan can be readily used as a "backend" analysis engine to enhance the existing static checkers such as BugFinder, PMD, or Coverity. We have performed a thorough evaluation of Graspan on five programs including the subjects written in C/C++—Linux kernel, the PostgreSQL database, the Apache httpd server, and Java subjects—the Hadoop Distributed File System (i.e., HDFS) and the Hadoop MapReduce (i.e., Hadoop-MapReduce). Our experiments show very promising results:

(1) **Scalability.** The three client analyses (i.e., context-sensitive field-insensitive pointer/alias analysis, context-sensitive field-sensitive pointer/alias analysis and context-sensitive dataflow analysis) running on top of Graspan scale easily to these systems, while their traditional implementations crashed in the early stage;
(2) **Efficiency**. Having the GPUs enabled, our GPU-based version Graspan-G managed to achieve orders of magnitude speedup against Graspan-C;
(3) **Development Effort**. In terms of lines of code, the Graspan-based implementations of these analyses are an order of magnitude simpler than their traditional implementations;
(4) **Effectiveness.** Using the results of these interprocedural analyses, the static checkers in Reference [89] have uncovered a total of **85** potential bugs and **1,308** unnecessary NULL tests in Linux, PostgreSQL and httpd. In addition, four Java bug checkers implemented by Graspan have discovered totally **103** potential bugs in HDFS and Hadoop-MapReduce.

1.5 Outline

The rest of this article is organized as follows: Section 2 introduces the background information regarding the graph reachability-based analysis. The "systemized" solution is then discussed in Section 3. In Section 4, we elaborate the programming model of Graspan, followed by the detailed descriptions of system design and implementation in Section 5. The evaluations are presented in Section 6. Finally, we discuss the related work (Section 7) and conclude the article (Section 8).

2 BACKGROUND

While there are many types of interprocedural analyses, this article focuses on a pointer/alias analysis and a dataflow analysis, both of which are enablers for all other static analyses. This section discusses necessary background information on how pointer/alias analysis is formulated as graph reachability problems. Following Reps et al.'s **interprocedural, finite, distributive, subset (IFDS)** framework [96], we have also formulated a fully context-sensitive dataflow analysis as a grammar-guided reachability problem. However, due to space limitations, the discussion of this formulation is omitted.

2.1 Graph Reachability

Pioneered by Reps et al. [96, 105], there is a large body of work on graph reachability-based program analyses [18, 60, 92, 121, 136, 138, 146, 148]. The reachability computation is often guided by a context-free grammar due to the *balanced parentheses* property in these analyses. At a high level, let us suppose each edge is labeled either an open parenthesis '(' or a close parenthesis ')'. A vertex is reachable from another vertex if and only if there exists a path between them, the string of labels on which has balanced '(' and ')'.

The parentheses '(' and ')' have different semantics for different analyses. For example, for a C pointer analysis, '(' represents an address-of operation & and ')' represents a dereference*. A pointer variable can point to an object if there is an assignment path between them that has

balanced & and *. For instance, a string "&&**" has balanced parentheses while "&**&" does not. This balanced parentheses property can often be captured by a context-free grammar.

2.2 Pointer Analysis

A pointer analysis computes, for each pointer variable, a set of heap objects (represented by allocation sites) that can flow to the variable. This set of objects is referred to as the variable's *points-to* set. Alias information can be derived from this analysis—if the points-to sets of two variables have a non-empty intersection, then they may alias.

Our graph formulation of pointer analysis is adapted from a previous formulation in Reference [153]. This section briefly describes this formulation. The analysis we implement is *flow-insensitive* in the sense that we do not consider control flow in the program. A program consists of a set of pointer assignments. Assignments can execute in any order, any number of times.

Pointer Analysis as Graph Reachability. For simplicity of presentation, the discussion here focuses on four kinds of three-address statements (which are statements that have at most three operands):

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

Complicated statements are often broken down into these three-address statements in the compilation process by introducing temporary variables. As with other analysis implementations, arithmetic computation conducted on pointers is not considered. To simplify discussion, we do not distinguish fields in a struct. That is, an expression $a \rightarrow f$ is handled in the same way as $*a$, with offset f being ignored. Field sensitivity can be easily added by following treatments in References [116, 117, 136].

For each function, an *expression graph*—whose vertices represent C expressions (including pointer variables, dereference expressions, address-of expressions, and malloc invocations) and edges represent value flow between expressions—is generated; graphs for different functions are eventually connected to form a whole-program expression graph. Each vertex on the graph represents an expression, and each edge is of three kinds:

- *Dereference edge (D)*: for each dereference $*x$, there is a D-edge from x to $*x$; there is also an edge from an address-of expression $\&x$ to x, because x is a dereference of $\&x$.
- *Assignment edge (A)*: for each assignment $x = y$, there is an A-edge from y to x; x and y can be arbitrary expressions.
- *Alloc edge (M)*: for each assignment $x = malloc()$, there is an M-edge from a special Alloc vertex to x.

Figure 1 shows a simple program and its expression graph. Each edge has a label, indicating its type. Solid and dashed edges are original edges in the graph and they are labeled M, A, or D,

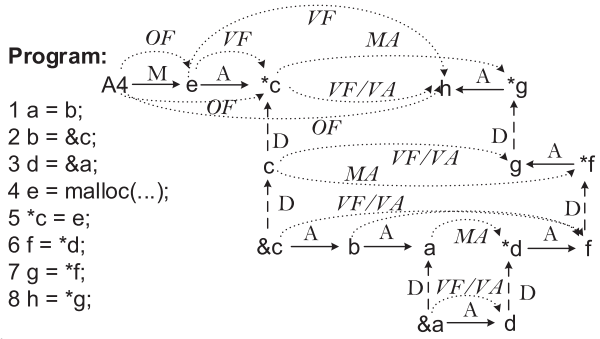


Fig. 1. A program and its expression graph: solid, horizontal edges represent assignments (A- and M- edges); dashed, vertical edges represent dereferences (D-edge); dotted, horizontal edges represent transitive edges labeled non-terminals. *A4* indicates the allocation site at Line 4.

respectively. Dotted edges are transitive edges[2] added by Graspan into the graph, as discussed shortly.

Context-free Grammar. The pointer information computation is guided by the following grammar:

$$Object\ flow: \quad OF ::= M\ VF \tag{6}$$

$$Value\ flow: \quad VF ::= (A\ MA?)^* \tag{7}$$

$$Memory\ alias: \quad MA ::= \overline{D}\ VA\ D \tag{8}$$

$$Value\ alias: \quad VA ::= \overline{VF}\ MA?\ VF \tag{9}$$

This grammar has four non-terminals *OF*, *VF*, *MA*, and *VA*. For a non-terminal T, a path in the graph is called a T-path if the sequence of the edge labels on the path is a string that can be reduced to T. For a variable v to point to an object o (i.e., a malloc), there must exist an *OF* path in the expression graph from o to v. 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.

There are two kinds of aliasing relationships in C: memory aliasing (MA) and value aliasing (VA). Two lvalue expressions are memory aliases if they may denote the same memory location while they are value aliases if they may evaluate to the same value.

An *MA* path is represented by $\overline{D}\ VA\ D$. Each edge has an inverse edge with a "bar" label. For example, for each edge $x \xrightarrow{D} y$, the edge $y \xrightarrow{\overline{D}} x$ exists automatically. $\overline{D}$ represents the inverse of a dereference and is essentially equivalent to an address-of. $\overline{D}\ VA\ D$ represents the fact that if (1) we take the address of a variable x and writes it into a variable y, (2) y is a *value alias of* another variable z, and (3) we perform dereferencing on z, the result is the same as the value in x.

A *VA* path is represented by $\overline{VF}\ MA?\ VF$. This has the meaning that if (1) two variables x and y are memory aliases, and (2) the values of x and y are propagated to two other variables z and u, respectively, through two *VF* paths, z and u contain the same pointer value. In other words, the path—$z\ \overline{VF}\ x\ MA\ y\ VF\ u$—induces $z\ VA\ u$.

[2]We use term "transitive edges" to refer to the edges dynamically added to represent non-terminals rather than the transitivity of a relation.

Note that *MA*, *VA*, and *VF* mutually refer each other. This definition captures the recursive nature of a flow or alias path. In this grammar, $\overline{D}$ and D are the open and close parentheses that need to be balanced.

Example. In Figure 1, e points to *A4*, since the M edge between them forms an *OF* path. There is a *VF* path from $\&a$ to d, which is also a *VA* path (since *VA* includes *VF*). The *VA* path enables an *MA* path from a to $*d$ due to the balanced parentheses D and $\overline{D}$. This path then induces two additional *VF*/*VA* paths from b to f and from $\&c$ to f, which, in turn, contribute to the forming of the *VF*/*VA* path from c to g, making $*c$ and $*g$ memory aliases. Hence, there exists a *VF* path from e to h, which, together with the M edge at the beginning, forms an *OF* path from *A4* to h. This path indicates that h points to *A4*. The dotted edges in Figure 1 show these transitive edges.

2.3 Dataflow Analysis

Following Reps et al.'s **interprocedural, finite, distributive, subset (IFDS)** framework [96], we have also formulated a fully context-sensitive dataflow analysis as a grammar-guided reachability problem. Particularly, we adopt this dataflow analysis to track NULL value propagation. Under the IFDS formulation, each dataflow fact corresponds to one vertex in a program graph. The dataflow transfer function performed on dataflow facts is interpreted as the relation mapping (edges) between vertices. As such, a program graph termed as "exploded supergraph" is finally generated. Performing the dataflow analysis is equivalent to the reachability computation on the graph. Please refer to Reference [96] for more technical details.

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 Reference [96], which has been demonstrated [132] to fall short in answering many user queries.

3 "BIG DATA" SOLUTION

The traditional way to implement the interprocedural analysis is to maintain a worklist, each element of which is a pair of a newly discovered vertex and a stack simulating a pushdown automaton. The implementation loops over the worklist, iteratively retrieving vertices and processing their edges. The traditional implementation does not add any physical edges into the graph (due to the fear of memory blowup), but instead, it tracks path information using pushdown automata. When a CFL-reachable vertex is detected, the vertex is pushed into the worklist together with the sequence of the labels on the path leading to the vertex. When the vertex is popped off of the list, the information regarding the reachability from the source to the vertex is discarded.

This traditional approach has at least two significant drawbacks. First, it does not scale well when the analysis becomes more sophisticated or the program to be analyzed becomes larger. For example, when the analysis is made *context-sensitive*, the grammar needs to be augmented with the parentheses representing method entries/exists; the checking of the balanced property for these parentheses also needs to performed. Since the number of distinct calling contexts can be very large for real-world programs, naïvely traversing all paths is guaranteed to be not scalable in practice. As a result, various abstractions and tradeoffs [57, 114–116] have been employed, attempting to improve scalability at the cost of precision as well as implementation straightforwardness.

Second, the worklist-based model is notoriously difficult to parallelize, making it hard to fully utilize modern computing resources. Even if multiple traversals can be launched simultaneously, since none of these traversals add transitive edges into the program graph as they are being detected, every traversal performs path discovery completely independently, resulting in a great deal of wasted efforts.

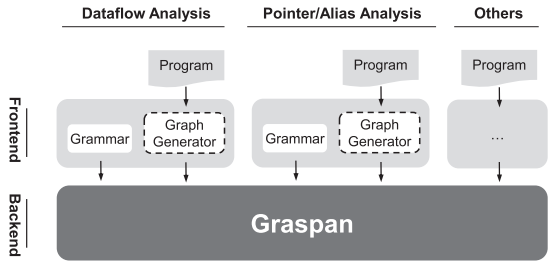


Fig. 2. Graspan usage; implementing a particular client analysis requires a specific frontend performing two tasks, namely, generating program graph and specifying analysis grammar; our Graspan system as the backend engine takes the generated program graphs and grammar rules as input and conducts analysis; users can choose either Graspan-C or Graspan-G according to the particular computing facility.

A "Big Data" Perspective. Our key insight here is that adding *physical* transitive edges into the program graph makes it possible to devise a Big Data solution to this static analysis problem for two reasons. First, representing transitive edges *explicitly* rather than *implicitly* leads to addition of a great number of edges (e.g., even larger than the number of edges in the original graph). This gives us a large (evolving) dataset to process. Second, the computation only needs to match the labels of consecutive edges with the productions in the grammar and is thus simple enough to be "systemized." Of course, dynamically adding many edges can make the computation quickly exhaust the main memory. However, this should not be a concern, as there are already many systems [44, 62, 73, 100, 124, 127] built to process very large graphs (e.g., the webgraph for the whole Internet).

4 GRASPAN'S PROGRAMMING MODEL

In this section, we describe Graspan's programming model, i.e., the tasks that need to be done by the programmer to use Graspan. Analogous to declarative program analysis [111, 140, 151], we separate the computation backend from the client-analysis implementations. Due to the general support for the efficient CFL-reachability computation Graspan performs, Graspan supports any client analyses that can be formulated as a CFL-reachability problem [95]. For users to implement a particular analysis, an analysis *frontend* is needed. As shown in Figure 2, we have implemented two frontends, one for dataflow analysis and a second for pointer/alias analysis. Users can customize their frontends for specific analyses, leaving the backend computation to Graspan. Note that, since frontends are separated from the backend analysis engine, users are freed from the burden of worrying about analysis performance or scalability. There are two main tasks for the user to build a frontend. The first task is to add a pass in a compiler infrastructure to generate the graph. The second task is to use the Graspan API to specify a grammar. Next, we will elaborate on these two tasks. Note that the programming model is applicable to both Graspan-C and Graspan-G. The differences between the two backends are transparent to users, who can compute with either of them with the same frontend.

Generating Graphs. For Graspan to perform an interprocedural analysis, the user first needs to generate the *Graspan graph*, which is a specialized program graph tailored for the analysis, by modifying a compiler frontend. Note that, since this task is relatively simple, the developer can generate the Graspan graph in a mechanical way without even thinking about performance and scalability. In this subsection, we briefly discuss how we generate the Graspan graph in the context of the pointer/alias analysis. We finish by generalizing graph generation for other interprocedural analyses.

For the pointer/alias analysis, we generate the Graspan graph by making two modifications to the program expression graph described in Section 2. These modifications include (1) inclusion of inverse edges and (2) context sensitivity achieved through inlining. For the former, we model inverse edges explicitly. That is, for each edge from a to b labeled X, we create and add to the graph an edge from b to a labeled $\overline{X}$.

For the latter, we perform a bottom-up (i.e., reverse-topological) traversal of the call graph of the program to inline functions. For each function, we make a *clone* of its entire expression graph for each call site that invokes the function. Formal and actual parameters are connected explicitly with edges. The cloning of a graph not only copies the edges and vertices in one function; it does so for *all* edges and vertices in its (direct and transitive) callees.

For recursive functions, we follow the standard treatment [132]—**strongly connected components (SCC)** are computed and then functions in each SCC are collapsed into one single function and treated context-insensitively. Clearly, the size of the graph grows exponentially as we make clones and the generated graph is often large. However, the out-of-core support in Graspan guarantees that Graspan can analyze even such large graphs effectively. For each copy of a vertex, we generate a unique ID in a way so we can easily locate the variable its corresponds to and its containing function from the ID. In the Graspan graph, edges carry data (i.e., their labels) but vertices do not. Finally, the graph is dumped to disk in the form of an edge list.

In general, the approach of aggressive inlining provides *complete information* that an analysis intends to uncover. Among all the existing analysis implementations, only Whaley et al. [132] could handle such aggressive inlining but they only clone variables (*not* objects) and have to use a **binary decision diagram (BDD)** to merge results. In addition, no evidence was shown that their analysis could process the Linux kernel. On the contrary, Graspan processes the exploded kernel graph in a few hours on a single machine.

Although this subsection focuses on the generation of pointer analysis graphs, graphs for other analyses can be generated in a similar manner. Here, we briefly summarize the steps. First, vertices and edges need to be defined based on a grammar; this step is analysis-specific. Second, if inverse edges are needed in the grammar, then they need to be explicitly added. Finally, context sensitivity can be generally achieved by function inlining. The developer can easily control the degree of context sensitivity by using different inlining criteria. For example, we perform *full context sensitivity* and thus our inlining goes from the bottom functions all the way up the top functions of the call graph. But if one wishes to perform only *one-level* context sensitivity, then each function only needs to be inlined once.

Specifying Grammar. Once the program graph is generated, the user needs to specify a grammar that guides the addition of transitive edges at runtime. Unlike any traditional implementation of the analysis, Graspan adds transitive edges (e.g., dotted edges in Figure 1) to the graph in a parallel manner. Specifically, for each production in the grammar, if Graspan finds a path whose edge labels match the RHS terms of the production, then a transitive edge is added covering the path and labeled with the LHS of the production.

Since Graspan uses the edge-pair-centric model, it focuses on a pair of edges at a time, which requires each production in the grammar to have no more than two terms on its RHS. In other words, the length of a path Graspan checks at a time must be ≤ 2.

For example, the above-mentioned pointer analysis grammar cannot be directly used, because the RHSes of *VF*, *MA*, and *VA* all have more than two terms. This means that to add a new *VF* edge, we may need to check more than two consecutive edges, which does not fit into Graspan's EP-centric model. Fortunately, every context-free grammar can be *normalized* into an equivalent grammar with at most two terms on its RHS [96], similar to the Chomsky normal form. After

normalization, our pointer analysis grammar becomes:

$$\begin{aligned}
&Object\ flow: && OF ::= M\ VF && (10)\\
&Temp: && T_1 ::= A \mid A\ MA && (11)\\
&Value\ flow: && VF ::= VF\ T_1 \mid \epsilon && (12)\\
&Memory\ alias: && MA ::= T_2\ D && (13)\\
&Temp: && T_2 ::= \overline{D}\ VA && (14)\\
&Value\ alias: && VA ::= T_3\ VF && (15)\\
&Temp: && T_3 ::= \overline{VF}\ MA \mid \overline{VF} && (16)
\end{aligned}$$

At the center of Graspan's programming model is an API:

`addConstraint(Label` *lhs*`, Label` *rhs1*`, Label` *rhs2*`)`,

which can be used by the developer to register each production in the grammar. *lhs* represents the LHS non-terminal, while *rhs1* and *rhs2* represent the two RHS terms. If the RHS has only one term, then *rhs2* should be NULL.

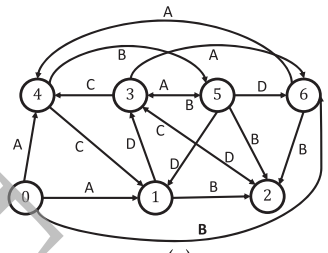
Graspan Applicability. How many interprocedural analyses can be powered by Graspan? First, we note that pointer analysis and IFDS-like dataflow analysis are already representatives of a large number of analysis algorithms that can be formulated as a grammar-guided graph reachability problem [95]. Second, work has been done to establish the convertibility from other types of analysis formulation (e.g., set-constraint [60] and pushdown systems [12, 13, 13]) to context-free language reachability. Analyses under these other formulations can all be parallelized and made scalable by Graspan.

Note that Graspan currently does not support analyses that require constraint solving, such as path-sensitive analysis and symbolic execution. Future work can add support for constraint-based analyses by encoding constraints into edge values such as Reference [158]. Two edges match if a satisfiable solution can be found for the conjunction of the constraints they carry. Moreover, Graspan currently only supports IFDS problems; we leave dataflow analyses and other flow-sensitive analyses that can not be formulated as IFDS problems to future work.

5 GRASPAN DESIGN AND IMPLEMENTATION

At the core of Graspan is an out-of-core algorithm for performing CFL reachability, which is illustrated by Algorithm 1. It consists of three main phases: preprocessing, computation, and postprocessing. A *preprocessing* step is first launched to divide the input graph into multiple partitions (Line 3). The main *analysis computation* is comprised of multiple *supersteps*. At each superstep (Lines 5–10), the scheduler is first invoked to decide which two partitions to load and process (Line 5). The system then loads the specified partitions into memory (Lines 6 and 7) and performs CFL-reachability computation on them (Line 8), which corresponds to Algorithm 3 or 4. At the end of the superstep, the updated partitions are written back to disks (Lines 9 and 10). The iterative process continues until it is terminated by the scheduling algorithm (Line 5)—no new edges can be generated. Finally, a *postprocessing step* is performed, after the computation, to parse and report the analysis results (Line 12).

We have implemented Algorithm 1 on both CPU and GPU. By taking into account the architectural differences between CPU and GPU, we devise distinct data representations and parallel operations to support CFG-reachability computation on those devices, resulting in Graspan-C (Section 5.3) and Graspan-G (Section 5.4). These backends share the same preprocessing (Section 5.1),



(a)

Partition 0

Src	Dst	Label
0	1	A
	4	A
	6	B
1	2	B
	3	D
2	3	C

Partition 1

Src	Dst	Label
3	2	D
	4	C
	5	B
	6	A
4	1	C
	5	B

Partition 2

Src	Dst	Label
5	1	D
	2	B
	3	A
	6	D
6	2	B
	4	A

(b)

Fig. 3. (a) An example graph and (b) its partitions.

ALGORITHM 1: An out-of-core algorithm for CFL-reachability computation

```
1  begin
2      /*Preprocessing*/
3      initializePartitions()
4      /*Iterative computation*/
5      for null ≠ ⟨i, j⟩ ← schedule() do
6          p_i ← load(i)                          // Disk->Memory
7          p_j ← load(j)                          // Disk->Memory
8          compute(p_i, p_j)
9          store(p_i)                             // Memory->Disk
10         store(p_j)                             // Memory->Disk
11     /*Postprocessing*/
12     postprocess()
```

scheduling (Section 5.2) and postprocessing (Section 5.6) steps, although the computing engine implementations are different.

5.1 Preprocessing

Preprocessing partitions the Graspan graph generated for an analysis. The graph is in the edge-list format on disk. Similar to graph sharding in GraphChi [62], partitioning in Graspan is done by first dividing vertices into *logical intervals*. However, unlike GraphChi, which groups edges based on their target vertices, one interval in Graspan defines a partition that contains edges whose *source vertices* fall into the interval. Edges are sorted on their source vertex IDs and those that have the same source are stored consecutively and *ordered on their target vertex IDs*. The fact that the outgoing edges for each vertex are sorted enables quick edge addition, as we will discuss shortly. Figure 3(a) shows a simple directed graph. Suppose Graspan splits its vertices into three intervals 0–2, 3–4, and 5–6; Figure 3(b) shows the partition layout.

When a new edge is found during processing, it is always added to the partition to which the source of the edge belongs. Graspan loads two partitions at a time and joins their edge-lists (Section 5.3 and Section 5.4), a process we refer to as a *superstep*. Given that only two partitions reside in memory at a given time, the size and hence the total number of partitions are determined automatically by the amount of memory available to Graspan.

Preprocessing also produces three pieces of meta-information: a *degree file* for each partition, which records the (incoming and outgoing) degrees of its vertices, a global ***vertex-interval table*** **(VIT)**, which specifies vertex intervals, and a ***destination distribution map*** **(DDM)** for each

partition p that maps, for each other partition q, the number of edge pairs potentially matched between p and q. DDM[p, q] is calculated by summing up the outgoing degree of the vertices in partition q that are out-neighbors of vertices in partition p. The DDM is essentially a matrix, with each cell containing a count.

Graspan uses the degree file to calculate the size of the array to be created to load a partition. Without the degree information, a variable-size data structure (e.g., ArrayList) has to be used, which would incur array resizing and data copying operations. The VIT records the lower and upper-bounds for each interval (e.g., (0, [0, 10000]), (1, [10001, 23451]), etc.). Graspan maintains the table because the intervals will be redefined upon repartitioning. The DDM measures the "matching" degree between two partitions and will be used by the Graspan scheduler to determine which two to load.

Note that Graspan supports both in-memory and out-of-core computations. For small graphs that can be held in memory, our preprocessing only generates two partitions, both of which are resident in memory. Since the VIT and the DDM are reasonably small in size, they are kept in memory throughout the processing.

5.2 Scheduling

When a new superstep starts, two new partitions will be selected by the scheduler to join. Since a partition on which the computation was done in the previous superstep may be chosen again, Graspan delays the writing of a partition back to disk until the new partitions are chosen by the scheduler. If a chosen partition is already in memory, then significant amounts of disk I/O can be saved.

ALGORITHM 2: DDM updating

```
for each partition p ∈ [0, n) do
    for each partition q ∈ [0, n) do
        if inMemory(p) & inMemory(q) then
            if repartition happened then
                compute the exact number of pairs potential matched
            else
                DDM[p, q] ← 0
        else if inMemory(p) & newEdges(p) & onDisk(q) then
            num ← calculate the number of edges in p whose target vertices belong to q
            DDM[p, q] ← num × totalEdges(q)/totalVertices(q)
        else if onDisk(p) & inMemory(q) & newEdges(q) then
            if repartitioning has happened on p then
                DDM[p, q] ← DDM[p, q]/numPartitions
        else if onDisk(p) & onDisk(q) then
            /*no updating is needed*/
```

The insight here is that regardless of the order in which supersteps are performed, the total number of added edges remains identical. To reduce the number of supersteps needed and thus the I/O cost, the partitions where more edges are likely to be added should be scheduled at a higher priority.

To this end, we develop a novel scheduling algorithm with two objectives: (1) maximize the number of edge pairs that can potentially match and (2) favor the reuse of in-memory partitions. For (1), the scheduler consults the DDM. As mentioned earlier, each cell of the DDM contains

the number of edge pairs that can match between two partitions. Our scheduler selects a pair of partitions that have the largest $DDM[p, q]+DDM[q, p]$ score. If multiple pairs of partitions have the same score (e.g., in a user-defined range), then Graspan picks the one that involves an in-memory partition.

At the beginning, the DDM is initialized during preprocessing. Given partition p and q, $DDM[p, q]$ is calculated as follows: For each edge $x \rightarrow y$ in partition p, if the target vertex y belongs to partition q, then the outgoing degree of vertex y in partition q is added to the value stored in $DDM[p, q]$. In this way, the initial value of each DDM cell indicates the maximum number of edges that can possibly match between two partitions. As the computation proceeds, edges in each partition are changed. The DDM has to be updated at the end of each superstep. Algorithm 2 lists the pseudo-code that updates the DDM. Essentially, for each partition pair of p and q, four cases need to be considered, depending on whether the partition is in memory or on disk.

- Both p and q are in memory: If the computation is completed, then $DDM[p, q]$ is set to zero; if the computation is interrupted due to repartitioning, then the DDM has to be recomputed.
- p is in memory with new edges added and q is on disk: In this case, we are not able to compute the exact number without loading q into memory. Thus, we estimate the value by multiplying the number of edges in p whose target vertices belong to q and the average out-degree of vertices in q.
- p is on disk and q is in memory with new edges added: If p is repartitioned, then we simply divide $DDM[p, q]$ by the number of partitions.
- Neither p nor q resides in memory: In this case, the current superstep has not performed any computation on p or q; hence, $DDM[p, q]$ remains the same.

5.3 Edge-pair-centric Computation on CPUs

We devised **edge-pair-centric (EP-centric)** computation model for iterative edge induction at each superstep. To accelerate the computation, we leveraged modern parallel computing facilities—multi-core CPUs and many-core GPUs—and developed two parallel versions, Graspan-C for CPUs and Graspan-G for GPUs. This section presents the design and implementation details of Graspan-C with regard to the in-memory data structure for graph representation and the parallel edge addition. We discuss Graspan-G shortly in Section 5.4.

In-memory Edge Representation. When two partitions are loaded into memory, we need to design an in-memory data structure for storing these partitions. For program analysis workloads, the input graphs are (1) large – the Linux kernel program graph has more than one billion edges; (2) sparse – its density is low; and (3) dynamic – edges are constantly added into graphs. The representation has to concisely represent the sparse graph with many edges and support efficient dynamic updates. As such, existing graph representations, such as adjacency matrix (tailored for dense graphs) or compressed row storage (for static graphs), are inadequate. In Graspan-C, we use an adjacency list to efficiently represent and manipulate a program graph.

The edge list of a vertex v is represented as two arrays of (vertex, label) pairs, as shown in Figure 4. The first array (O_v) contains "old" edges that have been inspected before and the second (D_v) contains edges newly added in the current iteration. The goal is to avoid repeatedly matching edge pairs (discussed shortly).

Parallel Edge Addition. Algorithm 3 shows a BSP-like algorithm for the parallel EP-centric computation. With two partitions p_1 and p_2 loaded, we first merge them into one single partition with combined edge lists (Lines 1–2). Initially, for each vertex v, its two arrays O_v and D_v are set to empty list and the original edge list of v, respectively (Line 4 and Line 5). The loop between Line 7 and Line 22 creates a separate thread to process each vertex v and its edge list, computing tran-

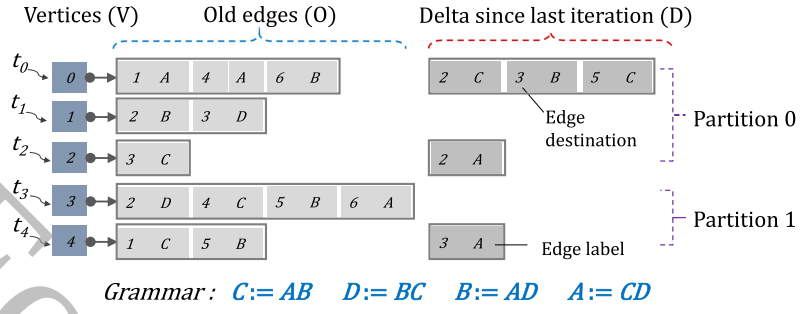


Fig. 4. The CPU in-memory representation of edge lists at the end of the first iteration.

sitive edges with two main components. After edge addition, both O_v and D_v are updated. The entire process is performed iteratively until the fixed point is reached—no new edges can be added anymore (i.e., D_v is empty for all vertices v).

The first component (Lines 9–15) attempts to match each "old" edge in O_v that goes to vertex u with each "new" edge of u in D_u. The second component (Lines 16–22) matches each "new" edge in D_v with both "old" and "new" edges in O_u and D_u of vertex u. The idea is that we do not need to match an "old" edge of v with an "old" edge of u, because this work has been done in a previous iteration. O_v and D_v are updated at the end of each iteration.

An important question is how to perform edge matching. A straightforward approach is that, for each edge $v \xrightarrow{L_1} u$, we inspect each of u's outgoing edges $u \xrightarrow{L_2} x$, and add an edge $v \xrightarrow{K} x$ if a production $K ::= L_1\ L_2$ exists. However, this simple approach suffers from significant practical limitations. First, before the edge is added into v's list, we need to scan v's outgoing edges one more time to check if the same edge already exists. Checking and avoiding duplicates is very important—duplicates may cause the analysis either not to terminate or to suffer from significant redundancies in both time and space.

Doing a linear scan of the existing edges is expensive—it has an $O(|E|^2)$ complexity to add edges for each vertex, where $|E|$ is the total number of edges loaded. An alternative is to implement an "offline" checking mechanism that removes duplicates when writing updated partitions to disk. While this approach eliminates the cost of online checks, it may prevent the computation from terminating—if the same edge is repeatedly added, missing the online check would make the loop at Line 7 keep seeing new edges and run indefinitely.

Our algorithm performs *quick edge addition* and *online duplicate checks*. Our key insight is that edge addition can be done *in batch* much more efficiently than *individually*. To illustrate, consider Figure 3(a) where vertex 0 initially has two outgoing edges $0 \rightarrow 1$ and $0 \rightarrow 4$. Adding new edges for vertex 0 is essentially the same as *merging* the (outgoing) edges of vertex 1 and 4 into vertex 0's edge list and then filtering out those that have mismatched labels.

In Algorithm 3, to add edges for vertex v, we first compute set V_1 by intersecting the set of target vertices of the edges in O_v and the set V of all vertices in the loaded partitions (Line 11). V_1 thus contains the vertices whose edge lists need to be merged with v's edge list. If an out-neighbor of v is not in V, then we skip it in the current superstep—this vertex will be processed later in a future superstep in which its partition is loaded together with v's partition.

Next, we add O_v into a list *listsToMerge* together with D_u of each vertex u in V_1 (Lines 10–13), and merge these lists into a new sorted list (Line 15). Since all input lists are already sorted, function MatchAndMergeSortedArrays can be efficiently implemented by repeatedly checking the grammar, finding the minimum (using an $O(log|V|)$ min-heap algorithm [15]) among the elements in a slice of the input lists and copying it into the output array. This whole algorithm has an

ALGORITHM 3: The parallel EP-centric computation on CPUs

```
   Input: Partition p1, Partition p2
1  Combine the vertices of p1 and p2 into V
2  Combine the edge lists of p1 and p2 into E
3  for each edge list v : (e1, e2, ..., en) ∈ E do in parallel
4      Set Ov to ()
5      Set Dv to (e1, e2, ..., en)
6  while there exists at least one vertex v whose Dv is NOT empty do
7      for each vertex v : (Ov, Dv) do in parallel
8          mergeResult_v ← ()
9          /*Merge Ov with only Dv of other vertices*/
10         List listsToMerge ← {Ov}
11         Let V1 be the intersection of the target vertices of Ov and V
12         foreach vertex u ∈ V1 do
13             Add Du into listsToMerge
14         /*Merge the sorted input lists into a new sorted list*/
15         mergeResult_v ← MatchAndMergeSortedArrays(listsToMerge)
16         /*Merge Dv with Ov ∪ Dv of other vertices*/
17         listsToMerge ← {Dv, mergeResult_v}
18         Let V2 be the intersection of the target vertices of Dv and V
19         foreach vertex u ∈ V2 do
20             Add Ou and Du into listsToMerge
21         /*Merge the sorted input lists into a new sorted list*/
22         mergeResult_v ← MatchAndMergeSortedArrays(listsToMerge)
23     for each vertex v : (Ov, Dv) do in parallel
24         /*Update Ov and Dv*/
25         Ov ← MergeSortedArrays(Ov, Dv)
26         Dv ← mergeResult_v − Ov
```

O($|E|log|V|$) complexity, which is more efficient, both theoretically and empirically, than scanning edges individually (O($|E|^2$)), because $|V|$ is much smaller than $|E|$. Furthermore, edge duplicate checking can be automatically done during the merging—if multiple elements have the same minimum value, then only one is copied into the output array. Label matching is performed before copying—an edge is not copied into the output if it has an inconsistent label.

Lines 16–22 perform the same logic by computing a new set of vertices V_2, and merging D_v and all the edges (i.e., $O_u \cup D_u$) of each vertex $u \in V_2$. At Line 22, all the new edges to be added to vertex v are in $mergeResult_v$. Finally, to prepare for the next iteration, O_v and D_v are merged (Line 25) to form a new O_v, which is then updated (Line 26) with the newly added edges (excluding those that already exist in O_v). We separate the computation (Lines 7–22) from the updates (Lines 23–26) to ensure that both O_v and D_v are *read* only during the parallel computation, avoiding data races.

Example. Figure 4 shows the in-memory edge lists at the end of the first iteration of the loop at Line 7 in Algorithm 3. In the next iteration, thread t_0 would merge O_0 with D_1 and D_4, and D_0 with $O_2 \cup D_2$ and $O_3 \cup D_3$. O_0 and O_1 (and O_4) do not need to be merged again as this has been done before.

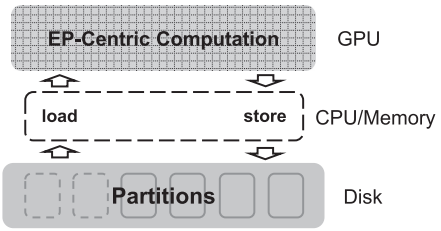


Fig. 5. The high-level architecture of Graspan-G.

Another advantage of this algorithm is that it runs completely in parallel without needing any synchronization. While the edge list of a vertex may be *read* by different threads, edge addition can only be done by one single thread, that is, the one that processes the vertex.

When a superstep is done, the updated edge lists need to be written back to their partition files. In addition, the degree file is updated with the new vertex degree information. The (in-memory) DDM needs to be updated with the new edge distribution information.

5.4 Edge-pair-centric Computation on GPUs

We additionally devise a GPU-based backend to accelerate closure computation on machines equipped with GPUs. The high-level architecture of Graspan-G is shown in Figure 5. Its workflow follows Algorithm 1, where in each superstep two partitions are loaded from the disk through the main memory to a GPU. Once the computation finishes on the GPU, the partitions are written back to the disk through the main memory. The supersteps iterate until the scheduler terminates the computation. In this design, edge induction is shifted to the GPU, while the CPUs are responsible for other components including preprocessing, loading/storing, and scheduling. In the following, we first discuss the necessary background about the microarchitecture of GPUs and then elaborate the EP-centric model on a GPU.

GPU Architecture. We take the NVIDIA GPU as an example to briefly describe the microarchitecture of modern graphics processors for general purpose computation. A typical general-purpose GPU consists of multiple identical computation units called ***streaming multiprocessors* (SMs)**. Each SM contains the instruction unit for instruction fetching, multiple **processing elements (PE)** for parallel execution, and the shared memory (mostly an L1 cache) for data exchange among threads running on the SM. Within an SM, each PE is able to run an independent thread. One SM often contains 32 PEs. Thus, a logical group of 32 threads—that constitute a *warp*—can run simultaneously on one SM. To use a GPU, a multi-thread program is partitioned into blocks of threads that are distributed to SMs. Since PEs on one SM have to share a single instruction unit, they must either execute the same instruction on individual data items (one per cycle) or wait. If threads in a warp execute different instructions, e.g., branch divergence occurs, the warp is automatically subdivided by the hardware into sets of threads executing the same instruction. These sets are then serially executed until reconvergence, which often leads to performance degradation.

Another feature that significantly affects performance is memory coalescing. When threads in a warp simultaneously access words (in the main memory) that are aligned in a single 128-byte segment, the hardware coalesces the 32 word accesses into one memory transaction that is as fast as accessing one word. However, if addresses scatter in memory, then multiple memory transactions have to be launched. In general, the more distributed the addresses are, the lower the throughput. Such hardware characteristics need to be considered in the design of Graspan-G. In particular, by

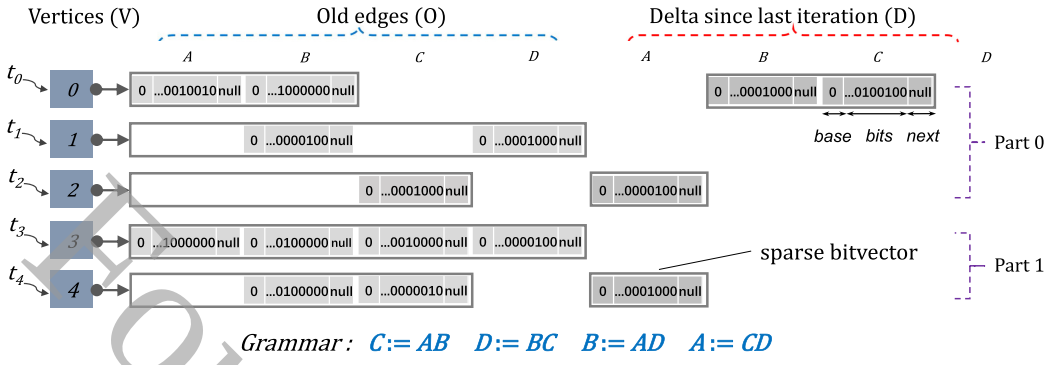


Fig. 6. The sparse bit vector representation of edge lists on GPUs at the end of the first iteration.

tailoring the in-memory data structure and edge addition algorithm for GPUs, Graspan-G minimizes branch divergence and increases memory throughput.

In-memory Edge Representation. Considering GPU's hardware characteristics, the graph representation discussed earlier for Graspan-C is no longer suitable. In Graspan-C, each source vertex is assigned to a thread responsible for adding new edges associated with that vertex. One naïve way to leverage this design on GPU is to distribute 32 source vertices to a warp (with 32 GPU threads), where each thread processes one source vertex (i.e., the whole computation indicated by Lines 7–22 in Algorithm 3). This is, however, inefficient, because (1) the control flow within each thread frequently diverges and (2) the edge lists to be merged scatter in memory. An alternative is to let each merge task (i.e., function MatchAndMergeSortedArrays) executed by a warp. Although this approach can increase memory throughput, it does not reduce the high frequency of branch divergence caused by duplicate checks and label matching.

To efficiently use GPU, we need a data structure tailored for its parallelism model. The data structure should better support fine-grained parallelization with minimized control-flow divergence. To this end, we propose to use a *bit vector* to represent an edge list. Moreover, different from the CPU-based edge representation where edge IDs and labels are stored together, we separate IDs and labels in this GPU-based representation. For each vertex, its edges are stored in a set of bit vectors, each of which contains edges with the same label. In Figure 6, the grammar has four labels, and hence each vertex has at most four bit vectors, each corresponding to a label. By putting the edges with the same label in the same bit vector, we are able to merge these edges efficiently by performing bit-wise union between vectors, effectively avoiding explicit label checking that leads to divergence. With this data structure, each warp is given a task that merges multiple bit vectors via bit-wise operations, as discussed shortly in Algorithm 4.

Although bit vectors are suitable for GPU's parallelism model, they generally have low space efficiency especially for sparse graphs (i.e., most elements of a vector are zero). This not only wastes the GPU cycles, but also introduces significant I/O in our out-of-core design. To tackle this problem, we use a sparsity-friendly variant, called *sparse bit vector* [78], that is designed specifically to represent sparse data. Internally, a sparse bit vector is a linked list of bit vectors, each element of which consists of three fields: *base* (indicates the range of integers possibly contained in the current element), *bits* (indicates whether a particular integer belongs to the set or not), and *next* (a pointer to the next element).

Each sparse bit vector uses 32 words (i.e., 128 bytes)—*base* and *next* are of one-word length each, while the *bits* part spans 30 words. The assignment of parallel tasks is done in a warp-centric manner. Each vertex is assigned to a warp. The sparse bit vector associated with the vertex is

ALGORITHM 4: The parallel EP-centric computation on GPUs

Input: Partition p_1, Partition p_2

1 Combine the vertices of p_1 and p_2 into V
2 Combine the edge lists of p_1 and p_2 into E
3 **for** *each edge list* $v : (e_1, e_2, \ldots, e_n) \in E$ **do in parallel**
4 Set O_v to ()
5 Set D_v to $(e_1, e_2, \ldots, e_n)$
6 **while** *there exists at least one vertex* v *whose* D_v *is NOT empty* **do**
7 **for** *each vertex* $v : (O_v, D_v)$ **do in parallel**
8 BitVector $mergeResult_v \leftarrow ()$
9 /*Merge O_v with only D of other vertices*/
10 **foreach** *each vertex* $w \in O_v.A$ **do** /*each edge v->w labeled with A in O_v*/
11 **if** *the rule* $\boldsymbol{C ::= A\ B}$ *exists* **then**
12 /*bitwise union two bitvectors in parallel by a GPU warp*/
13 $mergeResult_v.C \leftarrow parallel\_bitwise\_union(mergeResult_v.C,\ D_w.B)$
14 /*Merge D_v with $O \cup D$ of other vertices*/
15 **foreach** *each vertex* $w \in D_v.A$ **do** /*each edge v->w labeled with A in D_v*/
16 **if** *the rule* $\boldsymbol{C ::= A\ B}$ *exists* **then**
17 /*bitwise union bitvectors in parallel by a GPU warp*/
18 $mergeResult_v.C \leftarrow parallel\_bitwise\_union(mergeResult_v.C,\ O_w.B,\ D_w.B)$
19 **for** *each vertex* $v : (O_v, D_v)$ **do in parallel**
20 /*Update O_v and D_v in parallel by a GPU warp*/
21 $O_v \leftarrow parallel\_bitwise\_union(O_v,\ D_v)$
22 $D_v \leftarrow parallel\_bitwise\_diff\,(mergeResult_v,\ O_v)$

accessed and processed by a warp. The 32-word width fits the GPU architecture well. Loading one element of the sparse bit vector from the main memory requires only one memory transaction. Each of the 30 threads (except the first and last thread) in a warp performs the same operation over each word in the *bits* field, achieving high thread convergence.

Since a graph often does not exhibit good spatial locality, we further relax the restriction of the 32-word length requirement by using a variable-length sparse bit vector whose element size can be adjusted to be 2^i words based on the number of edges each vertex has in our program graph. Each vertex is then assigned to a *virtual warp* [49] for parallel processing, which provides increased flexibility and efficiency.

Parallel Edge Addition. At the heart of Graspan-G is a GPU-based algorithm for parallel edge addition. Briefly, under the sparse bit vector data structure, Graspan-G assigns to each warp the task of merging multiple bit vectors, which can be efficiently implemented via the bit-wise union. Label matching is done before parallel merging, while duplicate checking is enabled by the union operation.

Algorithm 4 describes the GPU version of the EP-centric computation. Graspan-G loads two partitions into GPU's memory and merges them into a single partition with combined edges lists (Lines 1–2), similarly to Algorithm 3. For each vertex, we create the "old" (O_v) and "delta" (D_v) vectors. The "old" vector is set empty, while the "delta" is initialized to be the original edge list (Lines 4 and 5). Next, we assign each vertex a virtual warp. During the processing of each vertex

v, a temporary bit vector $mergeResult_v$ is first instantiated to maintain the newly generated edges (Line 8). We induce edges in two steps, where the first step attempts to merge the old vector O_v with the delta vectors D of other vertices (Lines 10–13), and the second step merges the delta vector D_v with both the old and delta vectors $O \cup D$ of other vertices (Lines 15–18).

In each step, we apply all grammar rules for each vertex. For example, given a vertex v and a rule **C ::= A B**, we traverse the set of A-neighbors of v (Line 10). The traversal performs decoding the bit vector representing the outgoing edge list. For each A-neighbor vertex, say w, a bit-wise union is performed by a warp over the bit vector containing all its B-neighbors (i.e., $D_w.B$) and the temporary bit vector (i.e., $mergeResult_v.C$) that stores the newly generated C-neighbors of v (Line 13). Similarly, the bit-wise union operation is also applied in the second step to generate new edges in parallel (Line 18). After the edge induction, both O_v and D_v are updated for the next iteration. O_v and D_v are unioned to become the new O_v for the next iteration (Line 21). We replace D_v with the newly generated bit vector $mergeResult_v$, excluding the old vector O_v (Line 22).

Note that a warp is in charge of the bit-wise union operation over two bit vectors. Due to the uniform computation logic and the special data layout designed for sequential accesses, Graspan-G achieves both low thread divergence and high memory throughput. Moreover, the bit-wise diff operation can be implemented efficiently based on the bit-wise union and bit-wise negation: $parallel\_bitwise\_diff(a[i], b[i]) = parallel\_bitwise\_union(a[i],\ parallel\_bitwise\_neg(b[i]))$.

5.5 Repartitioning

If the size of a partition exceeds a threshold (e.g., a parameter), then repartitioning occurs. It is easy for Graspan to repartition an oversized partition, since the edge lists are sorted. Graspan breaks the original vertex interval into two small intervals, and edges are automatically restructured. The goal is to have the two small vertex intervals to have similar numbers of edges, so the resulting partitions have similar sizes. The VIT needs to be updated with the new interval information. Repartitioning can also be triggered in the middle of a superstep if too many edges are added in the superstep and the size of the loaded partitions is close to the memory size.

5.6 Postprocessing

Graspan provides an API for the user to iterate over edges with a certain label. For example, for the pointer analysis, edges with the *OF* label indicate a points-to solution, while edges with the *MA* and *VA* label represent aliasing variables. Graspan also provides translation APIs that can be used to map vertices and edges back to variables and statements in the program.

6 EVALUATION

We have implemented the CPU backend Graspan-C in C++ with approximately 4K lines of code. Graspan-G is implemented with 1.2K lines of C++ together with around 1K lines of CUDA code, where the functionalities running on CPUs (e.g., preprocessing, graph reading/writing, and repartitioning) are written in C++ and the CUDA code runs GPU computation.

6.1 Experimental Setup

Subject Programs & Analyses. We selected five large-scale software systems, including Linux kernel,[3] PostgreSQL database, Apache httpd server in C/C++, and Apache HDFS, Apache Hadoop-MapReduce written in Java as our analysis subjects. This set covers programs of different languages and domains, demonstrating Graspan's broad generality. We implemented three context-sensitive interprocedural analyses: field-insensitive Andersen's inclusion-based pointer/alias

[3]We focus on the code under x86 architecture.

Table 2. Characteristics of Subject Programs: Programs Analyzed, their Versions, Numbers of Lines of Code, Descriptions, and Languages Used

Subject	Version	#LoC	Description	Language
Linux	4.4.0-rc5	16M	Operating system	C/C++
PostgreSQL	8.3.9	700K	Database system	C/C++
httpd	2.2.18	300K	Http web server	C/C++
HDFS	2.0.3	546K	Distributed file system	Java
Hadoop-MapReduce	2.7.5	568K	Data processing engine	Java

analysis, field-sensitive pointer/alias analysis, and dataflow analysis to track null-pointer propagation. As the null-pointer analysis is not useful for Java programs where null pointers are explicitly reported, we ran it only for C/C++ programs. For field-sensitive alias analysis, we only ran it for Java programs. In Graspan, full context-sensitivity is achieved by cloning function bodies for every single calling context [107]. Table 2 shows the detailed characteristics of our programs.

For programs in C/C++, we built our frontend based on LLVM Clang. Our graph generators for the pointer/alias and dataflow analysis have 1.2K and 800 lines of C++ code, respectively. We first performed the pointer analysis. The dataflow analysis was designed specifically to track NULL value propagation. It was built based on the pointer analysis, because it needs to query pointer analysis results when analyzing heap loads and stores. For Java programs, we wrote about 700 lines of Java code to develop a graph generator based on the Soot compiler infrastructure.[4] Our grammar was adapted from References [117, 136] for both field-insensitive and field-sensitive alias analyses. In the field-insensitive analysis, we treated all fields as the same; whereas in the field-sensitive analysis, we distinguished loads and stores if they are performed on different fields.

We used a pre-computed call graph to perform inlining. For C/C++ programs, their call graphs are generated by using an inclusion-based context-insensitive flow-insensitive pointer analysis with support for function pointers (available in LLVM). The Java call graph is generated based on the Spark context-insensitive points-to analysis [66] available in Soot.

Hardware and Software Environment. Since our goal is to enable developers to use Graspan on development machines, we ran Graspan on a Dell desktop, with a quad-core 3.6 GHZ Intel Xeon W-2123 CPU, 8 GB memory, and a Samsung 860 EVO 1 TB SSD, running Ubuntu 16.04 LTS. The GPUs used on the machine is an Nvidia GeForce GTX 1080Ti with 28 SMs, 3,584 CUDA cores in total, and 11 GB memory, supported by the CUDA Toolkit 10.2.

Research Questions. Our evaluation focuses on the understanding of the following four research questions:

- Q1: Can the analyses we implemented find new bugs in large-scale systems? (Section 6.2)
- Q2: How does Graspan perform in terms of time and space and how much does the GPU-version speed up the computation? (Section 6.3)
- Q3: How do Graspan-based analysis implementations compare with other analysis implementations in terms of development effort and performance? (Section 6.4)
- Q4: How does Graspan compare with other backend systems when processing analysis workloads? (Section 6.5)

Since our analyses have already achieved the highest level of context sensitivity, we did not compare their precision with that of existing analyses. The main goal of this evaluation is to

[4] https://sable.github.io/soot/.

```
void* probe_kthread_data(task_struct *task){
      void *data = NULL;
      probe_kernel_read(&data);

      /*data will be dereferenced
        after return.*/
      return data;
}

long probe_kernel_read(void *dst){
      if(...) {
        return -EFAULT;
      }
      return __probe_kernel_read(dst);
}
```

(a) NULL deref in kernel/kthread.c

```
#define page_private(page)((page)->private)

bool swap_count_continued(...){
      head=vmalloc_to_page(...);
      if(page_private(head)!= ...){
        ...
      }
}
page*vmalloc_to_page(...){
      page *page = NULL;
      if (!pgd_none(*pgd)){
        //...
      }
      return page;
}
```

(b) NULL deref in mm/swapfile.c

Fig. 7. Two representative bugs in the Linux kernel 4.4.0-rc5 that were missed by the baseline checkers.

(1) demonstrate the usefulness of these interprocedural analyses through the detection of new bugs and (2) show the efficiency and scalability of Graspan when performing such expensive analyses that would be extremely difficult to make scalable otherwise.

6.2 Effectiveness of Interprocedural Analyses

To understand the effectiveness of our interprocedural analyses, we re-implemented the seven static checkers listed in Table 1 in Clang. We used these existing checkers as the baseline to understand whether the combination of interprocedural pointer/alias and dataflow analyses are able to improve them in finding new bugs or reducing false positives (as described in Table 1 in Section 1). Note that our interprocedural analyses are not limited to these checkers; they can be used in a much broader context to find other types of bugs as well (e.g., data races, deadlocks). We would also like to evaluate our analyses on commercial static checkers such as Coverity and GrammaTech. Unfortunately, we could not obtain a license that allows us to publish the comparisons, and hence, we had to develop these checkers from scratch.

We have added a new interprocedural checker UNTest that aims to find unnecessary, over-protective NULL tests—tests on pointers that must have non-NULL values—before dereferencing these pointers. Although these checks are not bugs, they create unnecessary code-level basic blocks that prevent compiler from performing many optimizations such as common sub-expression elimination or copy propagation, leading to performance degradation. Hence, these checks should be removed for compiler to fully optimize the program. To this end, we implemented an interprocedural *Must-Not-NULL* dataflow analysis. At each NULL test point, the analysis checks if the pointer involved must not be NULL.

We manually checked *all bug reports* from both the baseline checkers and our analyses (except those reported by UNTest as described shortly) to determine whether a reported bug is a real bug. Since some of these checkers (such as Block, Range, and Lock) are specifically designed for Linux, Table 3 only reports information w.r.t. the Linux kernel. For checkers that check generic properties (i.e., Null and UNTest), we have also run them on the two other programs; their results are described later in this section.

For the first six baseline checkers that found many real bugs in older versions of the kernel (used by Reference [89] in 2011 to check Linux 2.6.x and by Chou et al. [30] in 2001 to check Linux 2.4.x), they could find only two real bugs in Linux 4.4.0-r5 (with the Size checker). This is

Table 3. Checkers Implemented, their Numbers of Bugs Reported by the Baseline Checkers (BL), and *New Bugs* Reported by Our Graspan Analyses (GR) on Top of the BL Checkers on the Linux Kernel 4.4.0-r5

Checker	BL(4.4.0)		GR(4.4.0)		BL(2.6.1)
	RE	FP	RE	FP	RE
Block	0	0	0	0	43
Null	20	20	+108	23	98
Free	14	14	+4	4	21
Range	1	1	0	0	11
Lock	15	15	+3	3	5
Size	25	23	+11	11	3
UNTest	N/A	N/A	+1127	0	N/A

RE shows total numbers of bugs reported while **FP** shows numbers of false positives determined manually; to provide a reference of how bugs evolve over the past decade, we include an additional section **BL(2.6.1)** with numbers of *true* bugs reported by the same checkers in 2011 on the Kernel version 2.6.1 from Reference [89].
UNTest is a new *interprocedural* checker we implemented to identify unnecessary NULL tests; "+" means new problems found.

not surprising, because they were designed to target very simple bug patterns. Given that many static bug checkers have been developed in the past decade (including both commercial and open source), it is likely that most of these simple bugs have been fixed in this (relatively) new version of Linux. For example, the Null checker detected most of the bugs in References [89] and [30]. In this current version, while it reported 20 potential bugs, a manual inspection confirmed that all of them were false positives.

Unnecessary NULL Tests. We used our interprocedural analyses to identify NULL tests (i.e., `if(p)`) in which the pointers checked *must not* be NULL. We have identified a total of 1,127 unnecessary NULL tests in Linux, 149 in PostgreSQL, 32 in httpd. These are over-protective actions in coding and may result in performance degradation. Because these warnings are too many to inspect manually, we took a sample of 100 warnings and found these tests are truly unnecessary. This is the *first time* that unnecessary NULL tests in the Linux kernel are identified and reported.

New Bugs Found. Our analyses reported 108 new NULL pointer dereference bugs in Linux, among which 23 are false positives. All of these 85 new bugs involve complicated value propagation logic that cannot be detected by intraprocedural checkers. Figure 7 shows two example bugs.

In Figure 7(a), function `probe_kthread_data` invokes `probe_kthread_read` to initialize pointer `data`. However, in `probe_kthread_read`, if a certain condition holds, then an error code (`-EFAULT`) is returned and the pointer never gets initialized. Function `probe_kthread_data` then returns `data` directly without any check and the pointer gets dereferenced immediately after the function returns to its caller. In Figure 7(b), `page_private` may dereference a NULL pointer, since function `vmalloc_to_page` may return NULL. This bug was missed by the baseline because the origin of the NULL value and the statement that dereferences it are in separate functions. These types of bugs can only be found by interprocedural analyses. In fact, we show these two bugs because they are relatively simple and easy to understand; most of our bugs involve more than three functions and more complicated logic.

Table 4. Statistics of Bugs Found by Our Checkers in Java Programs

Subjects	I/O misuse	lock misuse	socket misuse	except. mishandle	total
HDFS	1 (1)	1 (1)	4 (4)	43 (9)	49 (15)
Hadoop-MR	0 (1)	0	0	54 (9)	54 (10)

Numbers of false positives in parentheses.

Table 5. A Breakdown of New Linux Bugs Found by Our Analyses

Modules	NULL pointer derefs	Unnecessary NULL tests
arch	0	75
crypto	0	15
init	0	1
kernel	4 (2)	65
mm	3 (0)	84
security	0	78
block	6 (2)	31
fs	19 (3)	84
ipc	0	17
lib	0	39
net	10 (8)	269
sound	15 (5)	83
drivers	25 (3)	286
Total	108 (23)	1,127

Numbers of false positives in parentheses.

For PostgreSQL and httpd, we detected 33 and 14 new NULL pointer bugs; our manual validation did not find any false positives among them.

To further validate the effectiveness of Graspan in bug detection, in addition to the checkers shown in Table 1 for C/C++ program, we implemented four other bug checkers for Java programs, targeting typestate-related bugs (i.e., I/O misuse, lock misuse, and socket misuse) [38, 118] and exception mishandling bugs [143]. In Graspan, we formulated these checkers as finite-state property verification problems, and implemented each as a context-sensitive dataflow analysis [31]. We ran these checkers on HDFS and Hadoop-MapReduce to report warnings. We manually inspected all warnings reported to determine if it is a real bug or a false positive warning. Table 4 demonstrates the statistics of bugs discovered. For the Java I/O checker, one bug was found in HDFS due to missing of a *close*. Two false positives were reported due to lack of support for the *try-with-resource* language feature. The socket checker reported 8 warnings in total, of which 4 are real bugs. The other 4 false positives were reported due to either our lack of path-sensitivity or misrecognition of socket initialization. The exception-handler checker, which detects mishandling of exceptions, successfully identified more than 100 cases where explicitly thrown exceptions never have handlers. Meanwhile, 25 false warnings were reported because of our lack of path-sensitivity.

Linux Bug Breakdown. Table 5 lists the new bugs and NULL tests in Linux into modules. We make two observations on this breakdown. First, the code quality of the Linux kernel has been improved significantly over the past decade. Note that the bugs we found are all complicated bugs detected by our interprocedural analyses; the baseline checkers could not find any (shallow) bug in this version of the kernel. Second, consistent with the observations made in both References [30] and [89], drivers is still the directory that contains most (NULL Pointer) bugs. This is not

Table 6. Graspan Performance Reported are the Numbers of Vertices and Edges before (**IS**) and after (**PS**) being Processed by Graspan, Numbers of Partitions Eventually (**#P**), Numbers of Supersteps Taken (**#SS**), and Total Running Time (**T**)

(a) Field-Insensitive Pointer/Alias Analysis

Program	IS=(E,V)	PS=(E,V)	Graspan-C			Graspan-G		
			#P	#SS	T	#P	#SS	T
Linux	(249.5M, 52.9M)	(1.1B, 52.9M)	12	20	1.1 hrs	14	32	10.9 mins
PostgreSQL	(25.0M, 5.2M)	(862.2M, 5.2M)	8	9	4.4 hrs	8	10	3.3 mins
httpd	(8.2M, 1.7M)	(904.3M, 1.7M)	6	9	2.6 hrs	7	10	1.5 mins
HDFS	(10.2M, 5.3M)	(1.8B, 5.3M)	9	13	9.9 hrs	7	8	55 secs
Hadoop-MR	(41.9M, 20.5M)	(99.2M, 20.5M)	4	8	6.9 mins	5	6	2.0 mins

(b) Field-Sensitive Pointer/Alias Analysis

Program	IS=(E,V)	PS=(E,V)	Graspan-C			Graspan-G		
			#P	#SS	T	#P	#SS	T
HDFS	(13.5M, 7.2M)	(741.8M, 7.2M)	5	7	1.2 hrs	9	6	21.2 mins
Hadoop-MR	(10.4M, 6.2M)	(297.3M, 6.2M)	4	10	1.0 hrs	7	5	1.8 mins

(c) Dataflow Analysis

Program	IS=(E,V)	PS=(E,V)	Graspan-C			Graspan-G		
			#P	#SS	T	#P	#SS	T
Linux	(69.4M, 63.0M)	(137.5M, 63.0M)	10	26	50.9 mins	12	28	3.2 mins
PostgreSQL	(34.8M, 29.0M)	(56.1M, 29.0M)	4	11	12.7 mins	5	9	2.8 mins
httpd	(10.0M, 5.3M)	(19.3M, 5.3M)	2	1	60 secs	2	1	14 secs

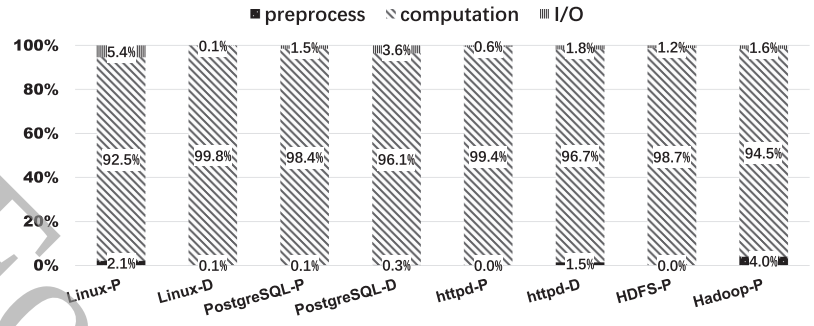
(a), (b), and (c) report the results for field-insensitive pointer/alias analysis, field-sensitive pointer/alias analysis, and dataflow analysis, respectively.

surprising, as drivers is still the largest module in the codebase. However, drivers is also the module of which developers are most cautious (perhaps due to the findings in References [30] and [89]), demonstrated by the most unnecessary NULL tests it contains.

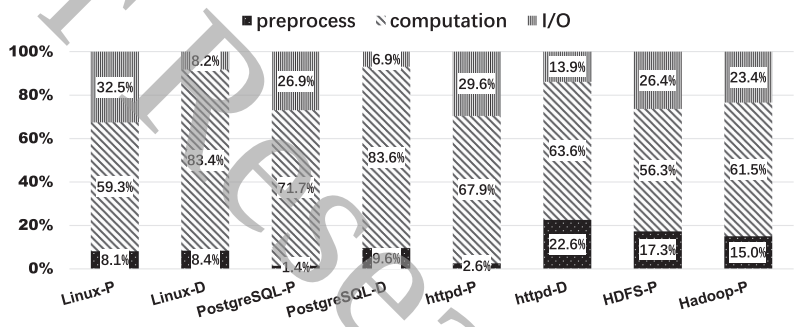
6.3 Graspan Performance

Overall Performance. Table 6 reports various statistics of Graspan's executions. Note that there is a large difference between the initial size and the post-processing size of each graph. For example, in Linux, the number of edges increases 3–5 times after the computation, while for httpd, the Graspan graph for pointer analysis increases more than 100 times. The computation time depends on both program characteristics and analysis type. For example, while the pointer analysis graph for `httpd` has a large number of edges added, its dataflow analysis graph does not change as much and thus Graspan-C finishes the computation quickly in 60 seconds. We found that this is because our dataflow analysis only tracks NULL values and in httpd the distances over which NULL can flow are often short.

We have also attempted to run these graphs *in memory* on the desktop we used, and all of them except the dataflow analysis of httpd ran out of memory. While the initial size of each graph is relatively small, when edges are added dynamically, the graph soon becomes very big and Graspan needs to repartition it many times to prevent the computation from running out of memory.



(a) Graspan-C



(b) Graspan-G

Fig. 8. The breakdown of Graspan's running time into preprocessing, computation, and I/O (i.e., disk reads/writes): (a) and (b) report results for Graspan-C and Graspan-G, respectively. P and D represent field-insensitive pointer/alias analysis and dataflow analysis.

Breakdown. Figure 8 reports the breakdown of Graspan's running time into preprocessing, computation, and I/O (i.e., disk writes/reads). For Graspan-C (Figure 8(a)), the EP-centric computation clearly dominates the execution. While Graspan leverages disk support, the I/O cost is generally low, because (1) the optimization is conducted to minimize the number of partition reads/writes; (2) most disk accesses are sequential accesses. The I/O time in pointer analysis of Linux (**Linux-P**) takes 5.4% of the total time, as it needs more supersteps for completion. Preprocessing is generally efficient; the highest value 4.0% comes from the pointer analysis on Hadoop-MapReduce (**Hadoop-P**) because Hadoop-P has a large input graph before computation.

Figure 8(b) demonstrates the performance breakdown of Graspan-G. Although the computation is still the largest contributor, its percentage decreases compared to that of Graspan-C. This can be easily understood, as the computation time is significantly reduced while other parts almost remain. In fact, the I/O time even slightly increases, as bit vectors may consume more space for sparse graphs. The time spent on preprocessing does not change much. Since the total running time decreases, the relative percentage of preprocessing becomes higher.

Graspan-C vs. Graspan-G. The **Speedup** section of Table 7 reports the speedups achieved by Graspan-G over Graspan-C on each analysis workload. With GPUs enabled, Graspan runs orders of magnitude faster. For example, for the field-insensitive pointer analysis of HDFS (**HDFS-P**), Graspan-G takes less than a minute to add more than a billion new edges; Graspan-C, in contrast, takes almost 10 hours.

Table 7. A Comparison on the Performance of Graspan, On-demand Pointer Analysis (ODA) [153] Implemented in Standard Ways, as Well as SociaLite [63] and Soufflé [56] Processing Our Program Graphs in Datalog, and Doop [20]

Analysis	Graspan-C	Graspan-G	Speedup	ODA [153]	SociaLite [63]	Soufflé [56]	Doop [20]
Linux-P	68.4 mins	10.9 mins	6.3	OOM	OOM	OOM	-
PostgreSQL-P	265.3 mins	3.3 mins	80.4	> 1 day	OOM	OOM	-
httpd-P	156.0 mins	1.5 mins	104.0	> 1 day	> 1 day	OOM	-
HDFS-P	594.8 mins	55 secs	648.9	-	OOM	OOM	OOM
Hadoop-P	6.9 mins	2.0 mins	3.5	-	> 1 day	32 secs	OOM
Linux-D	50.9 mins	3.2 mins	15.9	-	OOM	2.2 mins	-
PostgreSQL-D	12.7 mins	2.8 mins	4.5	-	OOM	41 secs	-
httpd-D	60 secs	14 secs	4.3	-	4 hrs	11 secs	-

P and D represent field-insensitive pointer/alias analysis and dataflow analysis, respectively. OOM means out of memory.

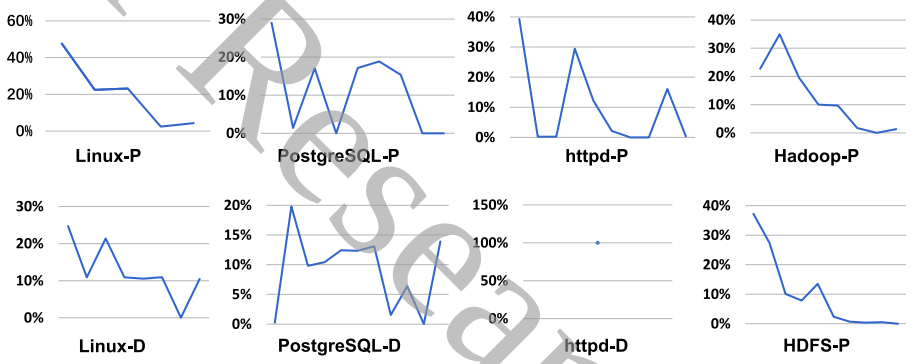


Fig. 9. Percentages of added edges across supersteps. P and D represent field-insensitive pointer/alias analysis and dataflow analysis, respectively. For the dataflow analysis of httpd (httpd-D), only one superstep is needed; 100% edges are added within the first superstep.

Edges Added. Figure 9 depicts the percentages of added edges across supersteps, measured as the number of added edges divided by the number of edges in each original graph for Graspan-C. In general, an extremely large number of edges are added within the first 10 supersteps (e.g., more than 500M for Linux), and as the computation progresses, fewer edges are added.

6.4 Comparisons with Other Analysis Implementations

Data Structure Analysis [65]. To understand whether Graspan-based analyses are more scalable and efficient than traditional analysis implementations, we wanted to compare our analyses with existing context-sensitive pointer/alias and dataflow analyses. While we had spent much time looking for publicly available implementations, we could not find anything available except the **data-structure analysis (DSA)** [65] in LLVM itself. DSA (implemented in 2007) is much more complicated than our pointer/alias analysis implementation—it has more than 10K lines of code, while our pointer/alias analysis (i.e., the graph generation part) only has 1.2K lines of code. According to a response from the LLVM mailing list [7], DSA was buggy and removed from LLVM since version 3.3. We tried to use LLVM 3.2, but it could not compile any version of the Linux kernel due to lack of patches.

On-demand Pointer Analysis [153]. As no other implementations were available, we implemented the context-sensitive version of Zheng and Rugina's C pointer analysis [153] ourselves. We took the expression graph generated by our frontend and used a worklist-based algorithm to

compute transitive closures, following closely the original algorithm described in Reference [153]. The *ODA* section of Table 7 reports its performance. For all but httpd, ODA either ran out of memory or took a very long time (longer than one day) on the same desktop where we ran Graspan. For example, when processing Linux, it ran out of memory in 13 minutes. When we moved it onto a server with 32 2.60 GHZ Xeon(R) processors and 32 GB memory, it took this implementation 3.5 days to analyze Linux and it consumed 29 GB out of the 32 GB memory. On the contrary, Graspan finished processing Linux in a few hours with less than 6 GB memory on the desktop with a much less powerful CPU.

Doop [20]. Doop is an analysis framework supporting k-limited context-sensitive pointer analysis for Java programs. It analyzes a program by translating it into relational data facts and leveraging a Datalog engine to perform the analysis. We ran a 2-callsite-context-sensitive analysis on HDFS and Hadoop-MapReduce using Doop. As shown in the last column of Table 7, Doop quickly crashed due to out-of-memory errors. This confirms with the observation made in the program analysis community: Memory is one of the major bottlenecks for scaling an analysis to large programs [10].

6.5 Comparisons with Other Backend Engines

Datalog. Since Datalog has been used to power static analyses, it is important to understand the pros/cons of using Graspan vs. a Datalog engine as the analysis backend. While there are many Datalog engines available [6, 63, 111, 125], SociaLite [63], LogicBlox [6], and Soufflé [8] are designed for shared-memory machines, while others [111, 125] are distributed engines running on large clusters. Since a distributed engine is often not a choice for code checking in daily development tasks, we focused our comparison against shared-memory engines. LogicBlox is a commercial tool that has been previously used to power the Doop pointer analysis framework [20] for Java. However, it was the same licensing issue that prevented us from publishing comparison results with LogicBlox. Hence, this subsection compares Graspan only with SociaLite, an early Datalog engine developed by Stanford, as well as Soufflé [8], the Datalog engine of choice for program analysis researchers.

The *SociaLite* section of Table 7 reports SociaLite's performance on the same desktop. SociaLite programs were easy to write—it took us less than 50 LoC to implement either analysis. However, SociaLite clearly could not scale to graphs that cannot fit into memory. For both pointer/alias and dataflow analysis, it ran out of memory for Linux and PostgreSQL. For httpd, although SociaLite processed the graphs successfully, it was much slower than Graspan.

The *Soufflé* section of Table 7 reports the performance of Soufflé under the same configuration. Soufflé failed to complete pointer analysis for almost all the subjects. For small analysis workloads where the program data can fit into memory, it outperforms Graspan. This is reasonable, since Graspan is designed for scaling large-scale analysis workloads. Its out-of-core design inherently introduces extra costs, e.g., preprocessing, partitioning, and disk I/O.

GraphChi. To understand whether other graph systems can efficiently process the same (program analysis) workload, we ran GraphChi—a disk-based graph processing system—because GraphChi is the only available system that supports both out-of-core computation and dynamic edge addition. GraphChi provides an API `add_edge` for the developer to add an edge; it maintains a buffer for newly added edges during computation and uses a threshold to prevent the buffer from growing aggressively. When the size of the buffer exceeds the threshold, the edge adding thread goes to sleep and the function always returns false. The thread periodically wakes up and checks whether the main data processing thread comes to the commit point, at which the edges in the buffer can be flushed out to disk. GraphChi does not check edge duplicates, and thus its computation would never terminate on our workloads. We added a naïve support that checks, before an edge

is added, whether the same edge exists in the buffer. Note that this support does not solve the entire problem, because it only checks the buffer but duplicates may have been flushed to shards. Checking duplicates in shards would require a re-design of the whole system.

We ran GraphChi on the same desktop to process the Linux dataflow graph. GraphChi ran into assertion failures in 133 seconds with around 65M edges added. This is primarily because GraphChi was not designed for the program analysis workload that needs to add an extremely large number of edges (with many duplicates) dynamically.

Graph Systems on GPUs. We wanted to compare Graspan-G against existing GPU-based graph processing systems. However, we could not find a reasonable baseline, because (1) existing GPU-based systems (e.g., Medusa [154], TOTEM [42], GunRock [130]) focus on in-memory computation—they cannot process large program graphs that do not fit into the GPU memory; and (2) these systems are tailored to processing static graphs and can hardly support the dynamic transitive closure computation in program analysis workloads.

7 RELATED WORK

Static Bug Finding. Static analysis has been used extensively in the systems community to detect bugs [1, 10, 16, 17, 21, 24, 32, 33, 36–39, 46, 69, 70, 85, 89, 102, 103, 129, 139] and security vulnerabilities [23, 25, 55]. Engler et al. [37] use a set of nine checkers to empirically study bugs in OS kernels. Palix et al. [89] implemented the same checkers using Coccinelle [88]. Commercial static checkers [2–5] are also available for finding bugs and security problems. Most of these checkers are based on pattern matching. Despite their commendable bug finding efforts, false positive and negatives are inherent with these checkers.

Interprocedural analyses such as pointer and dataflow analysis can significantly improve the effectiveness of the checkers, but their implementations are often not scalable. There exists a body of work that makes program analysis declarative [20, 132]—analysis designers specify rules in Datalog and these rules are automatically translated into analysis implementations. However, the existing Datalog engines perform generic table joining and do not support disk-based computation on a single machine. While declarative analyses reduce the development effort, they still suffer from scalability issues. For example, although the pointer analysis from Whaley et al. [132] can scale to reasonably large Java programs (e.g., using BDD), it only clones pointer variables, not objects. Furthermore, there is no evidence that they can perform fully context-sensitive analyses on codebases as large as the Linux kernel on a commodity PC.

Grammar-guided Reachability. There is a large body of work that can be formulated as a **context-free language (CFL)** reachability problem. CFL-recognition was first studied by Yannakakis [140] for Datalog query evaluation. Work by Reps et al. [50, 93, 95–97] proposes to model realizable paths using a context-free language that treats method calls and returns as pairs of balanced parentheses. CFL-reachability can be used to formulate a variety of static analyses, such as polymorphic flow analysis [92], shape analysis [94], points-to and alias analysis [18, 26, 116, 117, 117, 121, 136, 138, 146, 148, 153], and information flow analysis [72]. The works in References [60, 61, 77] study the connection between CFL-reachability and set-constraints, show the similarity between the two problems, and provide implementation strategies for problems that can be formulated in this manner. Kodumal et al. [61] extend set constraints to express analyses involving one context-free and any number of regular reachability properties. CFL-reachability has also been investigated in the context of recursive state machines [13], streaming XML [12], and pushdown languages [14]. Recent work uses CFL-reachability to formulate pointer and alias analysis [18, 116, 117, 121, 136, 138, 146–148, 153] and specification inference [18].

Context-sensitive Analyses. Generally, there are two dominant approaches to context-sensitive interprocedural analysis: the *summary-based* approach and the *cloning-based* approach [107]. The summary-based approach constructs a summary (transfer) function for each procedure, and directly applies the summary to the specific inputs at the call site invoking the function. Wilson et al. [82, 133] proposes to construct *partial transfer functions* to represent function summary as input-output value pairs for each procedure. The output value of a summary function can be directly exploited when the identical input pattern is encountered again for the same procedure. However, as a massive number of states need to be maintained, its scalability is still severely limited by the huge space consumption. Reps et al. proposes the IFDS [96] and IDE [105] dataflow frameworks, which is a variant of the summary-based approach. They concentrate on a particular subclass of interprocedural analysis problems, called the ***interprocedural, finite, distributive, subset problems (IFDS problems)***. While these frameworks can be instantiated to solve many analysis problems, they require the dataflow transfer function to be distributive over the meet operator (usually set union or intersection). However, most sophisticated analysis problems do not have such a property, e.g., pointer/alias analysis. Although the summary-based approach is scalable to certain extent, it suffers from more drawbacks. First, due to the lack of explicit representation of calling contexts, the summary-based approach fails to provide complete analysis information for each particular context. To answer the query like "to which memory locations does a pointer variable point under a particular context?" it has to recompute the pointer results along the calling context. Furthermore, it is difficult to precisely model heap effects. The cloning-based approach [34, 132] provides complete information. However, it requires each procedure to be re-analyzed under each calling context and hence is hard to scale.

Over the past decade, a body of work has been done to adjust the level of context sensitivity [52, 53, 67, 68, 74, 80, 114, 115, 135, 137, 149], explore different forms of sensitivity [57, 81], or leverage pre-processing analysis [87, 109, 113] to find sweet spots between scalability, generality, and usefulness. These techniques are largely orthogonal with Graspan, which can use them to perform *selective inlining* when generating graphs.

Parallel and Distributed Static Analyses. Researchers have proposed parallel and distributed static analysis algorithms and tools for increased efficiency and scalability. Mendez-Lojo et al. [79] propose a parallel points-to analysis based on constraint graph rewriting. A follow-up work [78] proposes a GPU-based algorithm. Nagaraj and Govindarajan [84] extend the graph-rewriting formulation and propose a parallel algorithm for the staged flow-sensitive pointer analysis. By leveraging data sharing and query scheduling, Su et al. [120] develop a parallel CFL-based pointer analysis to avoid redundant graph traversals. Their follow-up work [119] devises an algorithm for the heterogeneous CPU-GPU environment. Zhao et al. [150] attempt to parallelize an interprocedural flow-sensitive points-to analysis in the traditional task-parallel manner. Rodriguez et al. [99] propose an actor model-based parallel algorithm for dataflow analysis. Albarghouthi et al. [11] parallelize the demand-driven top-down interprocedural analysis with MapReduce. Garbervetsky et al. [41] propose a distributed program analysis framework on the basis of actor model where they implement a call-graph analysis. Blaß and Philippsen [19] describe an approach for parallelizing interprocedural dataflow analysis efficiently on a GPU. Google uses the distributed static analyses to analyze their large-scale codebase [104]. However, due to the challenges of performing interprocedural analysis on large-scale codebases, only simple (intra-procedural) analyses are applied. INFER [27] is a tool developed by Facebook based on bi-abduction to check memory safety properties for C code. For scalability, INFER only supports interprocedural analysis within each compilation unit.

Although these techniques were all designed for scalability, they suffer from several drawbacks. First, they were tailored for specific analysis algorithms and thus are not generally applicable.

Second, they all assume that memory is sufficient. However, as pointed out in Reference [10], memory is the major bottleneck for scaling analysis to large programs. Graspan aims to overcome these challenges, providing general support for sophisticated analysis algorithms without putting the burden of memory on analysis developers' shoulders.

Systems for Static Analyses. A recent line of work attempts to develop Big Data systems for scaling sophisticated static analysis. BDDBDDB [132] and Doop [20] are the early pioneers that run sophisticated static analysis on Datalog engines. These Datalog engines (even including a recent one *Soufflé* [56]) do not provide out-of-core disk support, and they are fundamentally limited by the size of main memory. None of them were able to scale a fully context-sensitive analysis to large-scale systems like Linux on the commodity desktop we used. Grapple [158] is a graph system designed for constraint-based path-sensitive static analysis. Chianina [159] is dedicated to scaling flow-sensitive analyses. BigSpa [45, 157] supports scalable context-sensitive analyses in a distributed setting. This work extends our previous work [126] that scales **context-free language (CFL)** reachability-based analyses to large programs with disk support. With GPUs enabled, our system achieves much higher efficiency than the original Graspan system [126] that processes program graphs only with CPUs.

Graph Systems. State-of-the-art graph systems include disk-based systems, shared-memory systems, distributed systems, and GPU-based systems.

Single-machine graph processing systems [9, 48, 62, 71, 75, 91, 101, 124, 127, 128, 152, 156] have recently become popular, because they enable big data to be processed locally, removing developers' burden of cluster management and maintenance. A wide set of single-machine systems were developed, including Ligra [112], Galois [86], GraphChi [62], X-Stream [101], GridGraph [156], GraphQ [127], MMap [71], FlashGraph [152], TurboGraph [48], Mosaic [75], RStream [128], and so on. Ligra [112], as a shared-memory system, is suitable for implementing parallel graph traversal algorithms. For graphs that fit in shared memory, Ligra shows significant performance advantages. Work from Reference [86] presents the design and implementation of the lightweight Galois infrastructure for graph analytics. By leveraging specially designed schedulers and data structures, Galois achieves excellent performance. GraphChi [62] introduces shards and proposes a parallel sliding algorithm to reduce disk I/O for out-of-core graph processing. To minimize random disk accesses, X-Stream [101] adopts an edge-centric model via streaming. GridGraph [156] adopts a grid representation for large-scale graphs by partitioning vertices and edges to 1D chunks and 2D blocks, respectively. Work from Reference [124] employs dynamic shards to reduce disk I/O. FlashGraph [152] implements a semi-external memory graph system that stores vertices in memory and edge-lists on SSDs. TurboGraph [48] manages adjacency lists in pages and leverages a cache to reduce disk I/O. Grapple [158], inspired by Graspan's design is a disk-based system designed for scalable constraint-based path-sensitive analysis of large programs.

Pregel [76], as the pioneering work of distributed graph systems, proposes a synchronous vertex-centric abstraction for large-scale graph processing. Following it, many other distributed systems [22, 28, 29, 43, 73, 83, 100, 108, 123, 134, 145, 155] have been developed based on the same graph-parallel abstraction. GraphLab [73] is a distributed framework for performing machine learning and data-mining algorithms on large-scale graphs. As an extension, PowerGraph [43] considers the structure of power-low graphs, thus achieving efficient graph partitioning and computation. Cyclops [28] supports synchronously computing over a distributed immutable view, granting a vertex with read-only access to all its neighboring vertices. Chaos [100] extends the streaming partitions introduced by X-Stream to multiple machines in a cluster and enables the parallel execution of streaming partitions. PowerLyra [29] is a graph system that differentiates the computation and partitioning on high-degree and low-degree vertices on skewed graphs. Gemini [155] is

a distributed system that adapts Ligra's hybrid push-pull computation model to a distributed form for computation-centric processing. CUBE [145] presents a new 3D task partitioning algorithm to reduce network traffic for certain machine learning and data mining applications. Built on top of Spark [144], GraphX [44] provides the high-level graph abstraction and "think like a vertex" interface for graph computation using low-level dataflow operators. Moreover, KickStarter [123] and Naiad [83] are the graph systems focusing on streaming graphs.

Thanks to the massive parallelism provided by a GPU, utilizing GPUs to accelerate graph processing has been shown to be a promising direction. Multiple GPU-based graph systems [40, 42, 54, 58, 59, 90, 106, 110, 130, 154] have been developed recently. Medusa [154] is a GPU-based graph-processing system providing a set of simplified programming interfaces. Since it requires loading the entire graph into the GPU memory, only small graphs that fit in the device memory can be processed. TOTEM [42] is a hybrid GPU-CPU system that divides a graph into two parts, assigned, respectively, to CPU and GPU for processing. CuSha [58] is a framework that supports the vertex-centric computation model. It exploits ***concatenated windows* (CW)** and g-shards graph representations to accelerate the computation. GunRock [130] adopts a bulk-synchronous and data-centric abstraction. It finds a sweet spot between performance and expressiveness by coupling high-performance GPU computing primitives with a high-level programming model. Frog [110] is a lightweight asynchronous processing framework with a preprocessing/hybrid coloring model. It employs a graph coloring algorithm to ensure that no adjacent vertices are divided into the same color-chunk. Vertices within each partition can be updated in parallel without modifying the data in adjacent vertices. In addition, several out-of-GPU-memory graph processing systems [47, 59, 106] are proposed to overcome the GPU memory capacity limitation, thus scaling to large-scale graphs.

Note that all of the above graph systems are designed for general-purpose graph applications mostly focusing on static graphs, while Graspan is a disk-based graph system tailored for dynamic transitive closure computation with broad applications in program analysis.

8 CONCLUSION

Graspan is the *first attempt* to turn sophisticated code analysis into scalable Big Data analytics, opening up a new direction for scaling various sophisticated static program analyses (e.g., symbolic execution, theorem proving) to large systems. To offer the performance flexibility, we develop two backends for Graspan, namely, *Graspan-C* running on CPUs and *Graspan-G* on GPUs. Graspan-C can analyze large-scale systems code on any commodity PC, while, if GPUs are available, then Graspan-G can be readily used to achieve orders of magnitude speedup by harnessing a GPU's massive parallelism.

REFERENCES

[1] 2015. The FindBugs Java Static Checker. Retrieved from http://findbugs.sourceforge.net/.
[2] 2016. The Coverity Code Checker. Retrieved from http://www.coverity.com/.
[3] 2016. The GrammaTech CodeSonar Static Checker. https://www.grammatech.com/codesonar-cc.
[4] 2016. The HP Fortify Static Checker. https://www.microfocus.com/en-us/cyberres/application-security/static-code-analyzer.
[5] 2016. The KlocWork Static Checker. https://www.perforce.com/products/klocwork.
[6] 2016. The LogicBlox Datalog Engine. Retrieved from http://www.logicblox.com/.
[7] 2016. Personal Communication with John Criswell.
[8] 2020. The *Soufflé* Datalog Engine. Retrieved from http://souffle-lang.github.io/.
[9] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. 2017. Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk I/O. In *USENIX ATC*. 125–137.
[10] Alex Aiken, Suhabe Bugrara, Isil Dillig, Thomas Dillig, Brian Hackett, and Peter Hawkins. 2007. An overview of the saturn project. In *PASTE*. 43–48.

[11] Aws Albarghouthi, Rahul Kumar, Aditya V. Nori, and Sriram K. Rajamani. 2012. Parallelizing top-down interprocedural analyses. In *PLDI*. ACM, 217–228. DOI: http://doi.org/10.1145/2254064.2254091
[12] Rajeev Alur. 2007. Marrying words and trees. In *PODS*. 233–242.
[13] Rajeev Alur, Michael Benedikt, Kousha Etessami, Patrice Godefroid, Thomas Reps, and Mihalis Yannakakis. 2005. Analysis of recursive state machines. *ACM Trans. Program. Lang. Syst.* 27, 4 (2005), 786–818.
[14] Rajeev Alur and P. Madhusudan. 2004. Visibly pushdown languages. In *STOC*. 202–211.
[15] M. D. Atkinson, J.-R. Sack, N. Santoro, and T. Strothotte. 1986. Min-max heaps and generalized priority queues. *Commun. ACM* 29, 10 (1986), 996–1000.
[16] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. 2004. SLAM and static driver verifier: Technology transfer of formal methods inside microsoft. In *IFM*. 1–20.
[17] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. 2001. Automatic predicate abstraction of C programs. In *PLDI*. 203–213.
[18] Osbert Bastani, Saswat Anand, and Alex Aiken. 2015. Specification inference using context-free language reachability. In *POPL*. 553–566.
[19] Thorsten Blaß and Michael Philippsen. 2019. GPU-accelerated fixpoint algorithms for faster compiler analyses. In *CC*. Association for Computing Machinery, New York, NY, 122–134.
[20] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA*. 243–262.
[21] Fraser Brown, Andres Notzli, and Dawson Engler. 2016. How to build static checking systems using orders of magnitude less code. In *ASPLOS*. 143–157.
[22] Yingyi Bu, Vinayak Borkar, Jianfeng Jia, Michael J. Carey, and Tyson Condie. 2014. Pregelix: Big(Ger) graph analytics on a dataflow engine. *Proc. VLDB Endow.* 8, 2 (Oct. 2014), 161–172.
[23] Suhabe Bugrara and Alex Aiken. 2008. Verifying the safety of user pointer dereferences. In *IEEE S&P*. 325–338.
[24] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*. 209–224.
[25] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2006. EXE: Automatically generating inputs of death. In *CCS*. 322–335.
[26] Cheng Cai, Qirun Zhang, Zhiqiang Zuo, Khanh Nguyen, Guoqing Xu, and Zhendong Su. 2018. Calling-to-reference context translation via constraint-guided CFL-reachability. In *PLDI*. Association for Computing Machinery, New York, NY, 196–210. DOI: http://doi.org/10.1145/3192366.3192378
[27] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving fast with software verification. In *NASA Formal Methods*. Cham, 3–11.
[28] Rong Chen, Xin Ding, Peng Wang, Haibo Chen, Binyu Zang, and Haibing Guan. 2014. Computation and communication efficient graph processing with distributed immutable view. In *HPDC*. 215–226.
[29] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *EuroSys*. 1:1–1:15.
[30] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. 2001. An empirical study of operating systems errors. In *SOSP*. 73–88.
[31] Manuvir Das, Sorin Lerner, and Mark Seigle. 2002. ESP: Path-sensitive program verification in polynomial time. In *PLDI*. 57–68.
[32] Robert DeLine and Manuel Fähndrich. 2001. Enforcing high-level protocols in low-level software. In *PLDI*. 59–69.
[33] Nurit Dor, Stephen Adams, Manuvir Das, and Zhe Yang. 2004. Software validation via scalable path-sensitive value flow analysis. In *ISSTA*. 12–22.
[34] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. 1994. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI*. ACM, New York, NY, 242–256. DOI: http://doi.org/10.1145/178243.178264
[35] Dawson Engler. 2011. Making finite verification of raw C code easier than writing a test case. In *RV*.
[36] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. 2000. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*. 1–1.
[37] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP*. 57–72.
[38] Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2006. Effective typestate verification in the presence of aliasing. In *ISSTA*. 133–144.
[39] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. 1999. A theory of type qualifiers. In *PLDI*. 192–203.
[40] Zhisong Fu, Michael Personick, and Bryan Thompson. 2014. MapGraph: A high level API for fast development of high performance graph analytics on GPUs. In *GRADES*. Association for Computing Machinery, New York, NY, 1–6.

[41] Diego Garbervetsky, Edgardo Zoppi, and Benjamin Livshits. 2017. Toward full elasticity in distributed static analysis: The case of callgraph analysis. In *ESEC/FSE*. ACM, New York, NY, 442–453. DOI: http://doi.org/10.1145/3106237.3106261
[42] Abdullah Gharaibeh, Elizeu Santos-Neto, Lauro Beltrão Costa, and Matei Ripeanu. 2013. Efficient large-scale graph processing on hybrid CPU and GPU systems. *CoRR* abs/1312.3018 (2013).
[43] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI*. 17–30.
[44] 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*. 599–613.
[45] 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. DOI: http://doi.org/10.1109/TPDS.2020.3036190
[46] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. 2002. A system and language for building system-specific, static analyses. In *PLDI*. 69–82.
[47] Wei Han, Daniel Mawhirter, Bo Wu, and Matthew Buland. 2017. Graphie: Large-scale asynchronous graph traversals on just a GPU. In *PACT*. 233–245. DOI: http://doi.org/10.1109/PACT.2017.41
[48] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. 2013. TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC. In *KDD*. 77–85.
[49] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. 2011. Accelerating CUDA graph algorithms at maximum warp. In *PPoPP*. Association for Computing Machinery, New York, NY, 267–276. DOI: http://doi.org/10.1145/1941553.1941590
[50] Susan Horwitz, Thomas Reps, and Mooly Sagiv. 1995. Demand interprocedural dataflow analysis. In *FSE*. 104–115.
[51] G. F. Italiano. 1986. Amortized efficiency of a path retrieval data structure. *Theor. Comput. Sci.* 48, 2–3 (1986), 273–281.
[52] 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 (Oct. 2018). DOI: http://doi.org/10.1145/3276510
[53] Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. 2017. Data-driven context-sensitivity for points-to analysis. *Proc. ACM Program. Lang.* 1, OOPSLA (Oct. 2017). DOI: http://doi.org/10.1145/3133924
[54] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. 2017. A distributed multi-GPU system for fast graph processing. *Proc. VLDB Endow.* 11, 3 (Nov. 2017), 297–310.
[55] Rob Johnson and David Wagner. 2004. Finding user/kernel pointer bugs with type inference. In *USENIX Security*. 9–9.
[56] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On synthesis of program analyzers. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 422–430.
[57] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid context-sensitivity for points-to analysis. In *PLDI*. 423–434.
[58] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. 2014. CuSha: Vertex-centric graph processing on GPUs. In *HPDC*. Association for Computing Machinery, New York, NY, 239–252.
[59] Min-Soo Kim, Kyuhyeon An, Himchan Park, Hyunseok Seo, and Jinwook Kim. 2016. GTS: A fast and scalable graph processing method based on streaming topology to GPUs. In *SIGMOD*. Association for Computing Machinery, New York, NY, 447–461.
[60] John Kodumal and Alex Aiken. 2004. The set constraint/CFL reachability connection in practice. In *PLDI*. 207–218.
[61] John Kodumal and Alex Aiken. 2007. Regularly annotated set constraints. In *PLDI*. 331–341.
[62] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale graph computation on just a PC. In *OSDI*. 31–46.
[63] Monica S. Lam, Stephen Guo, and Jiwon Seo. 2013. SociaLite: Datalog extensions for efficient social network analysis. In *ICDE*. 278–289.
[64] Butler W. Lampson. 1983. Hints for computer system design. In *SOSP*. 33–48.
[65] Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *PLDI*. 278–289.
[66] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java points-to analysis using SPARK. In *CC*. Springer-Verlag, Berlin, 153–169.
[67] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Precision-guided context sensitivity for pointer analysis. *Proc. ACM Program. Lang.* 2, OOPSLA (Oct. 2018). DOI: http://doi.org/10.1145/3276511
[68] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Scalability-first pointer analysis with self-tuning context-sensitivity. In *ESEC/FSE*. ACM, New York, NY, 129–140. DOI: http://doi.org/10.1145/3236024.3236041
[69] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2004. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI*. 20–20.

[70] Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *FSE*. 306–315.
[71] Zhiyuan Lin, Minsuk Kahng, Kaeser Md. Sabrin, Duen Horng (Polo) Chau, Ho Lee, , and U. Kang. 2014. MMap: Fast billion-scale graph computation on a PC via memory mapping. In *BigData*. 159–164.
[72] Ying Liu and Ana Milanova. 2008. Static analysis for inference of explicit information flow. In *PASTE*. 50–56.
[73] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.* 5, 8 (2012), 716–727.
[74] Jingbo Lu and Jingling Xue. 2019. Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity. *Proc. ACM Program. Lang.* 3, OOPSLA (Oct. 2019). DOI: http://doi.org/10.1145/3360574
[75] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a trillion-edge graph on a single machine. In *EuroSys*. 527–543.
[76] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, Grzegorz Czajkowski, and Google Inc. 2010. Pregel: A system for large-scale graph processing. In *SIGMOD*. 135–146.
[77] David Melski and Thomas Reps. 2000. Interconvertibility of a class of set constraints and context-free-language reachability. *Theoret. Comput. Sci.* 248 (2000), 29–98.
[78] Mario Mendez-Lojo, Martin Burtscher, and Keshav Pingali. 2012. A GPU implementation of inclusion-based points-to analysis. In *PPoPP*. ACM, 107–116. DOI: http://doi.org/10.1145/2145816.2145831
[79] Mario Méndez-Lojo, Augustine Mathew, and Keshav Pingali. 2010. Parallel inclusion-based points-to analysis. In *OOPSLA10*. ACM, 428–443. DOI: http://doi.org/10.1145/1869459.1869495
[80] Matthew Might, Yannis Smaragdakis, and David Van Horn. 2010. Resolving and exploiting the k-CFA paradox: Illuminating functional vs. object-oriented program analysis. In *PLDI*. ACM, New York, NY, 305–315. DOI: http://doi.org/10.1145/1806596.1806631
[81] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.* 14, 1 (Jan. 2005), 1–41. DOI: http://doi.org/10.1145/1044834.1044835
[82] Brian R. Murphy and Monica S. Lam. 1999. Program analysis with partial transfer functions. In *PEPM*. ACM, New York, NY, 94–103. DOI: http://doi.org/10.1145/328690.328703
[83] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A timely dataflow system. In *SOSP*. 439–455.
[84] Vaivaswatha Nagaraj and R. Govindarajan. 2013. Parallel flow-sensitive pointer analysis by graph-rewriting. In *PACT*. IEEE Press, 19–28.
[85] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: Type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.* 27, 3 (2005), 477–526.
[86] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *SOSP*. 456–471.
[87] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. 2014. Selective context-sensitivity guided by impact pre-analysis. In *PLDI*. Association for Computing Machinery, New York, NY, 475–484.
[88] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. 2008. Documenting and automating collateral evolutions in linux device drivers. In *EuroSys*. 247–260.
[89] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. 2011. Faults in Linux: Ten years later. In *ASPLOS*. 305–318.
[90] Y. Pan, Y. Wang, Y. Wu, C. Yang, and J. D. Owens. 2017. Multi-GPU graph analytics. In *IPDPS*. 479–490.
[91] Roger Pearce, Maya Gokhale, and Nancy M. Amato. 2010. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *SC*. 1–11.
[92] J. Rehof and M. Fähndrich. 2001. Type-based flow analysis: From polymorphic subtyping to CFL-reachability. In *POPL*. 54–66.
[93] Thomas Reps. 1994. Solving demand versions of interprocedural analysis problems. In *CC*. 389–403.
[94] Tom Reps. 1995. Shape analysis as a generalized path problem. In *PEPM*. 1–11.
[95] Thomas Reps. 1998. Program analysis via graph reachability. *Inf. Softw. Technol.* 40, 11–12 (1998), 701–726.
[96] T. Reps, S. Horwitz, and M. Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *POPL*. 49–61.
[97] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. 1994. Speeding up slicing. In *FSE*. 11–20.
[98] Liam Roditty and Uri Zwick. 2004. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In *STOC*. 184–191.
[99] Jonathan Rodriguez and Ondřej Lhoták. 2011. Actor-based parallel dataflow analysis. In *CC/ETAPS*. 179–197.
[100] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. 2015. Chaos: Scale-out graph processing from secondary storage. In *SOSP*. 410–424.

[101] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: Edge-centric graph processing using streaming partitions. In *SOSP*. 472–488.
[102] Cindy Rubio-González, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. 2009. Error propagation analysis for file systems. In *PLDI*. 270–280.
[103] Cindy Rubio-González and Ben Liblit. 2011. Defective error/pointer interactions in the Linux kernel. In *ISSTA*. 111–121.
[104] 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 (Mar. 2018), 58–66. DOI: http://doi.org/10.1145/3188720
[105] Mooly Sagiv, Thomas Reps, and Susan Horwitz. 1996. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoret. Comput. Sci.* 167, 1–2 (1996), 131–170.
[106] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. 2015. GraphReduce: Processing large-scale graphs on accelerator-based systems. In *SC15*. Association for Computing Machinery, New York, NY. DOI: http://doi.org/10.1145/2807591.2807655
[107] M. Sharir and A. Pnueli. 1981. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, S. Muchnick and N. Jones (Eds.). Prentice Hall, 189–234.
[108] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. 2016. Fast and concurrent RDF queries with RDMA-based distributed graph exploration. In *USENIX ATC*. 317–332.
[109] 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*. ACM, 693–706. DOI: http://doi.org/10.1145/3192366.3192418
[110] X. Shi, X. Luo, J. Liang, P. Zhao, S. Di, B. He, and H. Jin. 2018. Frog: Asynchronous graph processing on GPU with hybrid coloring model. *IEEE Trans. Knowl. Data Eng.* 30, 1 (2018), 29–42.
[111] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. 2016. Big data analytics with datalog queries on spark. In *SIGMOD*. 1135–1149.
[112] Julian Shun and Guy E. Blelloch. 2013. Ligra: A lightweight graph processing framework for shared memory. In *PPoPP*. 135–146.
[113] Yannis Smaragdakis, George Balatsouras, and George Kastrinis. 2013. Set-based pre-processing for points-to analysis. In *OOPSLA*. ACM, New York, NY, 253–270. DOI: http://doi.org/10.1145/2509136.2509524
[114] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick your contexts well: Understanding object-sensitivity. In *POPL*. 17–30.
[115] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective analysis: Context-sensitivity, across the board. In *PLDI*. 485–495.
[116] Manu Sridharan and Rastislav Bodik. 2006. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*. 387–400.
[117] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodik. 2005. Demand-driven points-to analysis for Java. In *OOPSLA*. 59–76.
[118] R. E. Strom and S. Yemini. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.* 12, 1 (Jan. 1986), 157–171. DOI: http://doi.org/10.1109/TSE.1986.6312929
[119] Yu Su, Ding Ye, and Jingling Xue. 2013. Accelerating inclusion-based pointer analysis on heterogeneous CPU-GPU systems. In *HiPC*. 149–158. DOI: http://doi.org/10.1109/HiPC.2013.6799110
[120] Yu Su, Ding Ye, and Jingling Xue. 2014. Parallel pointer analysis with CFL-reachability. In *ICPP*. 451–460. DOI: http://doi.org/10.1109/ICPP.2014.54
[121] Hao Tang, Xiaoyin Wang, Lingming Zhang, Bing Xie, Lu Zhang, and Hong Mei. 2015. Summary-based context-sensitive data-dependence analysis in presence of callbacks. In *POPL*. 83–95.
[122] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2016. Synergistic analysis of evolving graphs. *ACM Trans. Archit. Code Optim.* 13, 4 (2016).
[123] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. KickStarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *ASPLOS*.
[124] Keval Vora, Guoqing Xu, and Rajiv Gupta. 2016. Load the edges you need: A generic I/O optimization for disk-based graph processing. In *USENIX ATC*. 507–522.
[125] Jingjing Wang, Magdalena Balazinska, and Daniel Halperin. 2015. Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines. *PVLDB* 8, 12 (2015), 1542–1553.
[126] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. 2017. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. In *ASPLOS*. 389–404. DOI: http://doi.org/10.1145/3037697.3037744
[127] Kai Wang, Guoqing Xu, Zhendong Su, and Yu David Liu. 2015. GraphQ: Graph query processing with abstraction refinement—programmable and budget-aware analytical queries over very large graphs on a single PC. In *USENIX ATC*. 387–401.

[128] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. 2018. RStream: Marrying relational algebra with streaming for efficient graph mining on a single machine. In *OSDI'18*. USENIX Association, 763–782.
[129] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. 2013. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *SOSP*. 260–275.
[130] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and John D. Owens. 2017. Gunrock: GPU graph analytics. *ACM Trans. Parallel Comput.* 4, 1 (Aug. 2017).
[131] Cathrin Weiss, Cindy Rubio-González, and Ben Liblit. 2015. Database-backed program analysis for scalable error propagation. In *ICSE*. 586–597.
[132] John Whaley and Monica Lam. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*. 131–144.
[133] Robert P. Wilson and Monica S. Lam. 1995. Efficient context-sensitive pointer analysis for C programs. In *PLDI*. ACM, New York, NY, 1–12. DOI: http://doi.org/10.1145/207110.207111
[134] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. 2015. GraM: Scaling graph computation to the trillions. In *SoCC*. 408–421.
[135] Guoqing Xu and Atanas Rountev. 2008. Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis. In *ISSTA*.
[136] Guoqing Xu, Atanas Rountev, and Manu Sridharan. 2009. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *ECOOP*. 98–122.
[137] Guoqing Xu, Dacong Yan, and Atanas Rountev. 2012. Static detection of loop-invariant data structures. In *ECOOP 2012 – Object-Oriented Programming*, James Noble (Ed.). Springer Berlin, 738–763.
[138] Dacong Yan, Guoqing Xu, and Atanas Rountev. 2011. Demand-driven context-sensitive alias analysis for Java. In *ISSTA*. 155–165.
[139] Junfeng Yang, Can Sar, and Dawson Engler. 2006. EXPLODE: A lightweight, general system for finding serious storage system errors. In *OSDI*. 10–10.
[140] Mihalis Yannakakis. 1990. Graph-theoretic methods in database theory. In *PODS*.
[141] Daniel M. Yellin. 1993. Speeding up dynamic transitive closure for bounded degree graphs. *Acta Inf.* 30, 4 (1993), 369–384.
[142] S. Yong, S. Horwitz, and T. Reps. 1999. Pointer analysis for programs with structures and casting. In *PLDI*. 91–103.
[143] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. 2014. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *OSDI*. USENIX Association, 249–265. Retrieved from https://www.usenix.org/conference/osdi14/technical-sessions/presentation/yuan.
[144] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*. USENIX Association, 2.
[145] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. 2016. Exploring the hidden dimension in graph processing. In *OSDI*. 285–300.
[146] Qirun Zhang, Michael R. Lyu, Hao Yuan, and Zhendong Su. 2013. Fast algorithms for Dyck-CFL-reachability with applications to alias analysis. In *PLDI*. 435–446.
[147] Qirun Zhang and Zhendong Su. 2017. Context-sensitive data dependence analysis via linear conjunctive language reachability. In *POPL*. 344–358.
[148] Qirun Zhang, Xiao Xiao, Charles Zhang, Hao Yuan, and Zhendong Su. 2014. Efficient subcubic alias analysis for C. In *OOPSLA*. 829–845.
[149] Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. 2014. On abstraction refinement for program analyses in datalog. In *PLDI*. ACM, New York, NY, 239–248. DOI: https://doi.org/10.1145/2594291.2594327
[150] Jisheng Zhao, Michael G. Burke, and Vivek Sarkar. 2018. Parallel sparse flow-sensitive points-to analysis. In *CC*. Association for Computing Machinery, New York, NY, 59–70.
[151] Yue Zhao, Guoyang Chen, Chunhua Liao, and Xipeng Shen. 2016. Towards ontology-based program analysis. In *ECOOP (Leibniz International Proceedings in Informatics (LIPIcs))*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.), Vol. 56. Dagstuhl, Germany, 26:1–26:25. DOI: http://doi.org/10.4230/LIPIcs.ECOOP.2016.26
[152] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S. Szalay. 2015. FlashGraph: Processing billion-node graphs on an array of commodity SSDs. In *FAST*. 45–58.
[153] Xin Zheng and Radu Rugina. 2008. Demand-driven alias analysis for C. In *POPL*. 197–208.
[154] Jianlong Zhong and Bingsheng He. 2014. Medusa: Simplified graph processing on GPUs. *IEEE Trans. Parallel Distrib. Syst.* 25, 6 (June 2014), 1543–1552. DOI: http://doi.org/10.1109/TPDS.2013.111

[155] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *OSDI*. 301–316.
[156] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large scale graph processing on a single machine using 2-level hierarchical partitioning. In *USENIX ATC*. 375–386.
[157] 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*.
[158] Zhiqiang Zuo, John Thorpe, Yifei Wang, Qiuhong Pan, Shenming Lu, Kai Wang, Harry Xu, Linzhang Wang, and Xuandong Li. 2019. Grapple: A graph system for static finite-state property checking of large-scale systems code. In *EuroSys*. ACM.
[159] 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*. Association for Computing Machinery, New York, NY. DOI: http://doi.org/10.1145/3453483.3454085

Received August 2020; revised February 2021; accepted May 2021