Chianina: An Evolving Graph System for Flow- and Context-Sensitive Analyses of Million Lines of C Code

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

Yiyu Zhang[*]
Nanjing University, China
dz1933033@smail.nju.edu.cn

Qiuhong Pan[*]
Nanjing University, China
mg1733048@smail.nju.edu.cn

Shenming Lu[*]
Nanjing University, China
mf1733041@smail.nju.edu.cn

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

Linzhang Wang[*]
Nanjing University, China
lzwang@nju.edu.cn

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

Guoqing Harry Xu
University of California, Los Angeles
harryxu@cs.ucla.edu

Abstract

Sophisticated static analysis techniques often have complicated implementations, much of which provides logic for *tuning and scaling* rather than *basic analysis functionalities*. This tight coupling of basic algorithms with special treatments for scalability makes an analysis implementation hard to (1) make correct, (2) understand/work with, and (3) reuse for other clients. This paper presents Chianina, a graph system we developed for fully context- and flow-sensitive analysis of large C programs. Chianina overcomes these challenges by allowing the developer to provide only the basic algorithm of an analysis and pushing the tuning/scaling work to the underlying system. Key to the success of Chianina is (1) an *evolving graph formulation* of flow sensitivity and (2) the leverage of *out-of-core, disk support* to deal with memory blowup resulting from context sensitivity. We implemented three context- and flow-sensitive analyses on top of Chianina and scaled them to large C programs like Linux (17M LoC) on a single commodity PC.

CCS Concepts: • **Computer systems organization → Special purpose systems**; Reliability; • **Theory of computation → Program analysis**.

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

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06...$15.00
https://doi.org/10.1145/3453483.3454085

Keywords: static analysis, graph processing, parallel computing

ACM Reference Format:
Zhiqiang Zuo, Yiyu Zhang, Qiuhong Pan, Shenming Lu, Yue Li, Linzhang Wang, Xuandong Li, and Guoqing Harry Xu. 2021. Chianina: An Evolving Graph System for Flow- and Context-Sensitive Analyses of Million Lines of C Code. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3453483.3454085

1 Introduction

Static analysis plays important roles in a wide spectrum of applications, including bug detection, compiler optimization, *etc.*. Static analysis algorithms that distinguish results based on various program properties (*e.g.*, calling contexts and control flow) are more useful than those that do not. For example, these precise algorithms can uncover many true bugs and report less false warnings. As a result, there is an everlasting interest in program analysis community to develop techniques that are context-sensitive [17, 29, 36, 38, 69, 70, 74], field-sensitive [4, 36, 59, 61], flow-sensitive [23, 24, 29, 52], or path-sensitive [2, 15, 57, 82].

Although these techniques are superior to their (context, field, flow, or path-) insensitive counterparts, their computation is much more expensive, requiring CPU and memory resources that a single machine may not be able to offer. Given the limited resources to them, it is hard for them to scale to programs with large codebases such as the Linux kernel. Prior work employs sophisticated treatments that tune the level of sensitivity [39, 42, 78] or explore different forms of sensitivity [32, 46, 58], to find sweatspots between scalability, generality, and usefulness. Despite their commendable efforts, these treatments are specific to the applications they are developed for and complicated to implement.

This paper is a quest driven by the following question: given an analysis algorithm — *in its simplest form* — can we run it efficiently over large programs without requiring any sophisticated treatment from developers? Achieving this goal possesses a number of advantages: (1) analysis development is significantly simplified — because a developer only writes the basic algorithm without worrying about performance, this enables developers without much training in PL to easily develop and experiment with analyses that used to be accessible only to experienced experts; and (2) porting an existing analysis for different clients is significantly simplified because the analysis implementation contains only the logic necessary to realize the basic functionality, *not* any complex tuning tasks.

Insight and Problem. This paper is inspired by a line of prior work [6, 67–69, 82] that piggybacks static analysis on databases or large-scale systems — an analysis is implemented by following only a few high-level interfaces while scaling is delegated to the underlying system, which makes it possible for the analysis to run on large programs by enlisting the humongous computing power provided by modern hardware. BDDBDDB [69] and Doop [6] are early examples where an analysis is expressed as a Datalog program, which is executed by a low-level BDD-based Datalog engine for scalability. Graspan [67] is a graph processing system that leverages disk support to scale CFL-reachability computation to large programs that cannot fit into the main memory. This line of work shifts the burden of tuning from developers' shoulders to underlying systems, enabling developers to enjoy both the implementation simplicity and the scalability provided by the underlying system.

Inspired by these techniques, this paper revisits the problem of scaling *context- and flow-sensitive* analyses from a *system perspective* — that is, we aim to develop system support for scaling the *simplest versions* of context- and flow-sensitive algorithms that developers can quickly implement by following interfaces. On the one hand, a context- and flow-sensitive analysis is arguably one of the most expensive analysis techniques because it needs to compute and maintain an analysis solution for each distinct program point under each distinct calling context. On the other hand, it enables *strong update* and produces ultra precise information at each statement. For example, it is known in the community [23, 37] that flow sensitivity is critical for a C pointer analysis to prune away spurious points-to relationships.

State of the Art. One category of prior work dealing with context sensitivity focuses on computing and applying symbolic summaries [11, 70, 72, 76], which corresponds to the *bottom-up* approach in Sharir and Pnueli's seminal work [56]. Summary-based approach, while scalable for certain cases, still suffers from drawbacks. First, it is hard for certain analyses (*e.g.*, pointer analysis) to establish a succinct summary for each function [2, 70]. Moreover, due to lack of explicit representation of contexts, it cannot answer queries such as what objects a variable points to under a particular call stack. Another category of work is to aggressively clone functions [36, 59, 69, 74], which corresponds to the *top-down approach* in [56]. Cloning-based techniques often use optimizations such as merging, reduction, *etc.*, to gain scalability.

Work that deals with flow sensitivity includes the classical IFDS [52] and IDE [55] frameworks, which turn a dataflow analysis into a graph reachability problem over an exploded graph representation of a program. These frameworks require a dataflow transfer function to be *distributive over the meet operator* (*e.g.*, set union or intersection). However, many problems do not have this property; pointer/alias analysis is such an example. To scale flow-sensitive pointer analysis, researchers employ *sparse analysis* [23, 24, 26] over an SSA-based def-use graph that allows pointer information to be propagated only between statements that define/use same pointers.

All of these analyses, except those implemented on top of frameworks such as IFDS and IDE, have complicated implementations. Designing such an analysis requires a *full-set solution* — from the basic analysis algorithm all the way down to special treatments for efficiency/scalability that depart significantly from the basic algorithm. Commonalities often exist between treatments for different analyses, but are hard to reuse due to the tight coupling between basic algorithms and scalability treatments. Clearly, it would remain difficult for these techniques to gain real-world popularity until (1) their implementation complexity can be significantly reduced and (2) general frameworks can be developed to support a wide variety of them (*e.g.*, as analogous to how Apache Spark provides a general data-parallel foundation for various data analytics and machine learning tasks).

Problem Formulation. This paper presents a *domain-specific graph system* dubbed Chianina, that supports easy development of *any* context- and flow-sensitive analysis (with a monotone transfer function) for C and that is powerful enough to scale the analysis to many millions of lines of code. Chianina makes analysis implementation *simple* and *general* — a variety of flow-sensitive analysis (*e.g.*, analyses of IDE, IFDS, pointer, alias, type, value, *etc.*) can be developed with hundreds lines of code. The developer only specifies dataflow facts and transfer functions, *in their basic form without any special treatment.* Tuning and scaling (*e.g.*, merging, exploiting similarities, reduction, *etc.*), which used to be tightly coupled with the analysis, now happen under the hood.

A system-level solution requires *simple, mechanized computation over very large datasets.* To this end, Chianina uses aggressive cloning to implement context sensitivity — a callee is cloned into each of its callers and cloning is done in a bottom-up fashion from each leaf node on the call graph to the `main` function (if it exists). Cloning *streamlines* the implementation of any context-sensitive analysis and makes the analysis *highly parallel* due to elimination of sharing (§2.2). Of course, aggressively cloning function bodies can blow up the memory usage; Chianina overcomes memory limitations by leveraging out-of-core disk support. Once cloning is done, we have a complete, context-sensitive-by-construction program representation for graph computation.

To deal with flow sensitivity, Chianina formulates a flow-sensitive analysis as a problem of *evolving graph processing* [27, 48, 65, 66]. An evolving graph contains a set of temporally-related *graph snapshots*, each capturing the set of vertices and edges of the graph at a certain point of time. For example, a social network graph such as Twitter constantly evolves. Analytics tasks such as finding popular users (*i.e.*, PageRank) are often performed on snapshots of the graph periodically and results from these tasks are analyzed to understand the evolution of the graph. Two consecutive snapshots often have large overlap on vertices and edges (*i.e.*, spatial and temporal locality), which can be exploited for efficiency. *This nature of evolving graph processing matches exactly the nature of a flow-sensitive analysis* — at each program point, (the most general form of) dataflow facts for variables in the program constitute a graph snapshot; consecutive snapshots, which are captured at consecutive program points, differ only in a small number of vertices and edges due to application of transfer function.

Our formulation makes an analysis *amenable to many optimization techniques* (*e.g.*, auto-parallelization, work-balancing, locality, *etc.*) available in the graph system community, tuning and scaling the analysis at a low level without needing any special treatment from the developer. In fact, many of the prior analysis-level treatments are essentially equivalent to certain system-level optimizations (*e.g.*, BDD-based merging is essentially locality-aware compression). By pushing the tuning effort down into the system, *every analysis* running atop can enjoy these low-level optimizations, while in the past each analysis only receives a small handful of special treatments tailored for itself.

Note that our work makes no contribution to the static analysis algorithms. Our major contribution is building a scalable system to support a wide variety of static analyses. By leveraging auto-parallelization and out-of-core support, Chianina liberates developers from the fear of memory explosion, enabling straightforward implementations and a high-degree of parallelism.

Summary of Results. To validate scalability and generality, we implemented, on top of Chianina, the fully *context- and flow-sensitive* (1) pointer/alias analysis, (2) null-pointer value flow analysis, and (3) instruction cache analysis. We analyzed five large-scale software systems: Linux, Firefox, PostgreSQL, OpenSSL and Httpd. Our results are promising: our alias analyses completed on the five systems (4 minutes – 20 hours) whereas their conventional counterparts (even without context sensitivity) quickly ran out of memory for large programs. Chianina's source code is publicly available on GitHub: https://github.com/Chianina-system.

2 Background and Overview

We present Chianina in the context of pointer/alias analysis, which is one of the most sophisticated and expensive analyses in the context- and flow-sensitive analysis family. This section first offers a gentle introduction to the basic algorithm for a context- and flow-sensitive pointer/alias analysis for C (§2.1). Next, we provide an overview of Chianina (§2.2).

2.1 Background

Alias Analysis as Graph Reachability. A flow-insensitive alias analysis can be easily formulated as a graph-reachability problem. There are a number of existing formulations, of which we use the program expression graph (PEG) [80] based representation as an example to illustrate how Chianina works. Note that Chianina is a general framework that does not tie to PEG; other program representations can be used in Chianina as well.

A PEG represents a program as a graph where each vertex corresponds to a pointer expression (*e.g.*, a reference variable x, a dereference expression $*x$, or an address-of expression $\&x$). Edges are added based upon the following rules for statements that involve pointer expressions.

Type	*Stmt*	*Edge*	
assignment	$x = y$	$x \xleftarrow{a} y$	(1)
store	$*x = y$	$*x \xleftarrow{a} y$	(2)
load	$x = *y$	$x \xleftarrow{a} *y$	(3)
address-of	$x = \&y$	$x \xleftarrow{a} \&y$	(4)

Each statement allocating heap memory (*e.g.*, $x = malloc()$) is treated the same way as an address-of statement — we add an edge $x \xleftarrow{a} \&O$ where O represents the allocation site. Moreover, *dereference edges (d)* are added (1) from each pointer variable x to $*x$ and (2) from $\&x$ to x.

Based on this graph representation, the alias analysis is formulated as a reachability problem guided by a *context-free language* $\mathcal{L}$ over an alphabet Σ (*i.e.*, the set of $\{a, d\}$ in the context of PEG). Given a PEG whose edges are labeled with elements of Σ, we say a vertex v is $\mathcal{L}$-reachable from another vertex w if there exists a path from v to w on the graph such that the string formed by concatenating edge labels on the path is a member of language $\mathcal{L}$ (*i.e.*, complying with L's grammar). A whole-program alias analysis determines all pairs of such vertices v and w such that w is $\mathcal{L}$-reachable from v, based on the following context-free grammar:

$$\textit{Value alias} \quad V ::= (M?\ \overline{a})^*\ M?\ (a\ M?)^* \tag{5}$$

$$\textit{Memory alias} \quad M ::= \overline{d}\ V\ d \tag{6}$$

The non-terminals V and M represent the *value-alias* and *memory-alias* relations, respectively. Each PEG is a bidirectional graph — for each edge $x \xrightarrow{a} y$ with label a, there exists an inverse edge $y \xrightarrow{\overline{a}} x$ automatically. Two pointer expressions are aliases if they are $\mathcal{V}$- or $\mathcal{M}$-reachable. At the heart of this formulation is finding paths whose edge labels exhibit "balanced-parenthesis" properties (*e.g.*, a and $\overline{a}$): if a pointer value goes from a variable x into a heap location h and later flows to another variable y from a heap location i, the two variables x and y are (pointer) aliases if the two heap locations h and i are (memory) aliases. Given that this formulation is well-known to the PL community, we omit a concrete example here to save space.

Flow-Sensitivity. Flow sensitivity is often achieved using the traditional monotone dataflow analysis framework [31, 33], which consists of the analysis domain, including operations to copy and combine domain elements, and the transfer functions over domain elements with respect to different types of statement in the control flow graph. In the context of a PEG-based alias analysis, a straightforward way to add flow sensitivity is to model each domain element as a separate PEG and the combination operator as the union of edge sets. Each transfer function *w.r.t.* a program statement takes an input PEG that captures the state of the program before the statement, and computes an output PEG by adding and deleting edges according to the semantics of the statement.

Next, a worklist-based algorithm iteratively applies the transfer function for each statement along the control-flow graph (CFG). In our setting, two elements $\mathcal{IN}_s$ and $\mathcal{OUT}_s$ are maintained for each statement s of the CFG, representing the incoming and outgoing PEGs, respectively. Each transfer function s computes a new PEG $\mathcal{OUT}_s$ by adding/deleting edges on $\mathcal{IN}_s$. At each control flow join point where a node s has multiple predecessors $p \in predecessors(s)$, the incoming graph $\mathcal{IN}_s$ of node s is the union of all graphs $\mathcal{OUT}_p$ of its predecessors. The algorithm keeps updating these graphs until seeing the global fixed point [30]. Each transfer function is characterized as addition (*i.e.*, *GEN*) or deletion (*i.e.*, *KILL*) of a set of edges based on the aforementioned formulation. The *GEN* set usually denotes the new assignment edge (labeled with a) added due to a statement. The *KILL* set contains edges that must be deleted due to updated assignments. These deletions enable strong update.

Graph Representation of Dataflow Facts. Relating the PEG-based formulation of a flow-sensitive pointer analysis to the traditional monotone dataflow framework, it is easy to see that our (semi-) lattice here is a partial-order set containing *all possible edges* over the (finite) set of all pointer expressions, the `meet` operation is the set union, and the bottom element $\perp$ is empty set $\emptyset$. We use the flow-sensitive pointer/alias analysis as an example because its lattice is much more complicated than that of other dataflow analyses (which is often a small set of single elements rather than a relation). However, this does not preclude similar graph representations of simple lattices — thinking of a single-element set as a special relation where each element is modeled as a pair (*i.e.*, edge) $\langle l, \circ \rangle$ ($\circ$ is a special placeholder element), any dataflow fact can be modeled as a relation with a graph representation. Of course, for problems whose lattice is a set of single elements, graphs for their dataflow facts have a special structure — all edges have $\circ$ as their target vertex.

Note that the PEG representation discussed above describes the *basic analysis algorithm* without any scalability treatments. Naïvely running this algorithm will be unscalable. Chianina provides scalability with graph optimizations and disk support.

2.2 Chianina Overview

Chianina consists of a C-based frontend and a language-independent backend (which can be readily used to analyze programs in other languages although this paper focuses on the C language). The frontend is a Clang-based intraprocedural compiler pass that analyzes each C function to produce a control flow graph (CFG) of the function where each vertex of the CFG (*i.e.*, a statement) contains an PEG representing the dataflow fact at the statement. The initial PEG for each statement just contains edges induced by the statement itself. The backend is a graph engine that performs iterative computation over the CFG to update PEGs associated with each statement. The CFG generation is generic and independent of client analysis, but the graph representing each dataflow fact (contained in each CFG vertex) is *client-specific* and needs to be provided by the developer. For our pointer/alias analysis, each dataflow fact is a PEG, which will grow/shrink as computation is performed by the backend.

Note that the developer can also customize the CFG structure generated for each function. For example, our analysis implementation actually generates a sparse def-use graph proposed in [23], which is more efficient than the general CFG. For generality, we will still use term CFG in the rest of the paper to refer to the graph representation.

Cloning for Context Sensitivity. Once the CFG for each function is generated, Chianina relies on a pre-computed call graph (*i.e.*, constructed by LLVM) to perform cloning for context sensitivity. The CFG for each function is cloned and incorporated into that of each of its callers by creating assignment edges to connect vertices representing formal and actual parameters. Cloning of a CFG includes cloning of each PEG contained in each of its vertices.

To handle recursion, we first identify the strongly connected components (SCCs) over the pre-computed call graph. Functions in each SCC are cloned **twice** and treated context insensitively afterwards. In other words, functions not in any SCC enjoy full context sensitivity while a 2-level call-chain sensitivity is used for those in SCCs.

It is important to note that although there exists a body of work on other types of context sensitivity, **cloning is the type most suitable for a system solution like Chianina**. This is because cloning *streamlines* a context-sensitive analysis by generating a humongous global CFG (GCFG) that is *context sensitive by construction.* It makes it easy not only to *mechanize* analysis implementations but also to make them

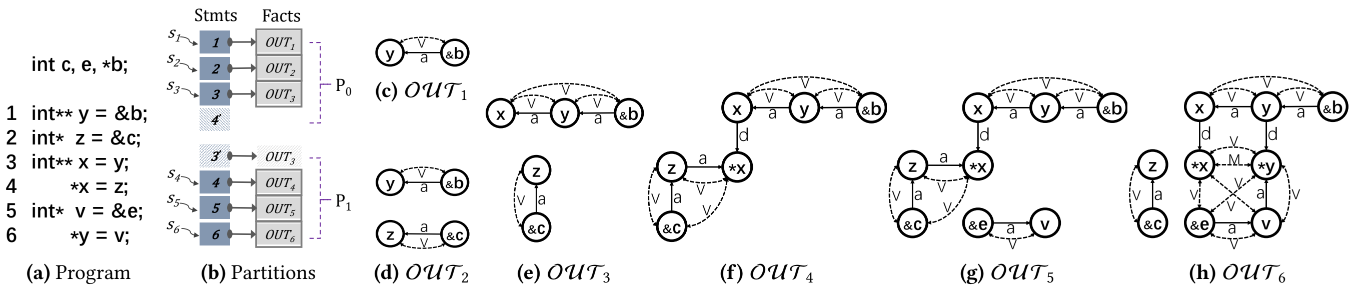


Figure 1. (a) The example program under analysis. (b) The two partitions: each CFG vertex links to a PEG; CFG edges are stored with their source vertices but not shown in the figure. (c)-(h) PEGs at each program point as iterative computation is performed by the backend graph engine; inverse edges are omitted for simplicity; The "V" and "M" edges represent transitive *value-alias* and *memory-alias* relationships shown earlier in Equations (5) and (6).

highly parallel as many threads can run the same analysis code over different parts of the graph *without any sharing*. As such, Chianina has near-linear thread-scalability, leading to superior performance (see §4.1). A high degree of parallelism requires (1) little sharing between threads and (2) overcoming memory limitations (because each thread needs to maintain its own analysis state and tracking data; running many threads thus requires large amounts of memory). Existing analysis implementations are limited by the size of main memory and hence cannot afford representing code separately for distinct contexts. As such, threads often have to work on a small program graph where code under different contexts is shared, leading to frequent synchronizations.

Evolving Graph Computation. Figure 1a shows an example C program. The dataflow fact associated with each statement, represented as a PEG, is initialized by the frontend compiler pass as a small PEG containing only edges induced by that statement. For space efficiency, only $\mathcal{OUT}$ is maintained explicitly since $\mathcal{IN}$ for a statement can be easily derived by taking a union of $\mathcal{OUT}$ of its predecessors.

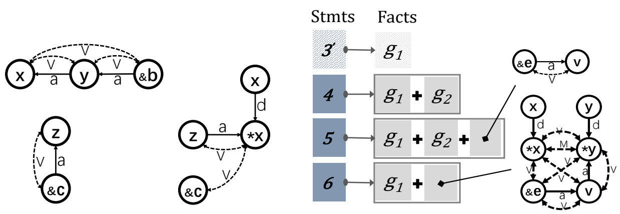
As the first step, Chianina divides the GCFG into multiple partitions. Figure 1b shows such an example with two disjoint partitions, containing vertices of the logical ranges [1-3] and [4-6], respectively. For edges that cross partitions, such as the one between statement 3 and 4 in Figure 1a, we create two *mirror vertices* 3′ and 4′ and place them respectively into the two partitions. Such edges induce dependencies between partitions. With multiple partitions available on disk, the Chianina scheduler picks a number of partitions at a time and loads them into memory for parallel computation. The number of partitions to load at each time is determined by (1) memory availability and (2) the number of CPU cores. Partitioning and scheduling is detailed in §3.3.

Assuming that both partitions are selected for computation in our example, Chianina loads into memory all CFG edges that belong to P_0 and P_1 and dataflow facts (PEGs) associated with each vertex. The computation engine runs the iterative algorithm over the subgraph represented by the partition in a *Bulk Synchronous Parallel* (BSP) style [44]. For our example, Chianina uses two threads to run the iterative computation over the two partitions. The iterative algorithm, which is the same as the traditional dataflow algorithm, keeps updating PEGs until a fixed point is reached. For example, when the computation reaches the mirror vertex 4 in P_0, it stops because vertex 4 is *not* present in the partition and there is no other path to continue the algorithm.

Before Chianina writes all updated PEGs back to disk for P_0, it adds statement 4 into the *active list* of P_1 via a message, together with the new PEG for this statement computed in P_0. When the current computation for P_1 finishes, the scheduler identifies that P_1 has an active vertex (meaning an updated PEG for the vertex has been computed from another partition). As a result, it selects P_1 for computation again in the next round. This next round of computation for P_1 is *incremental* — it starts at statement 4 (known as *frontier* in the terminology of graph processing) and only updates subsequent PEGs that are affected by the change. The repetitive process stops until a global fixed point is seen — no partition has any active vertices to process. In our example, the final $\mathcal{OUT}$ PEGs for the statements 1–6 are shown in Figure 1c–1h, respectively.

Alias Computation. There are two choices as to how to compute an alias solution (based on Equation 5 and 6) on each PEG. The first choice is that alias computation is performed on each PEG after the iterative algorithm finishes globally. While the approach simplifies the dataflow transfer function (which only needs to update direct assignment (*i.e.*, a-) edges during iterative computation), we are *not* able to perform *strong update* (*i.e.*, edge deletion) at each update because the pointer/alias information is unknown when transfer functions are applied. The second choice is we compute transitive edges on each PEG *on the fly* as the PEG is updated. This approach enables strong updates because the alias information is available at each update, at a cost of complicating transfer functions — now each transfer function has to additionally take care of addition/deletion of transitive (*i.e.*, V- and M-) edges besides assignment (a-) edges. Due to the importance of strong update in a flow-sensitive analysis, Chianina adopts the second approach, which computes and updates transitive edges on the fly.

To illustrate, consider statement 6 in Figure 1a where $*y$ points to a *singleton memory location*. A strong update is performed there — the effect of this is to *kill*, from the PEG $\mathcal{OUT}_5$, (1) all direct assignment edges going to $*y$ and expressions that *must alias* $*y$, as well as (2) all transitive edges induced by these assignment edges heretofore. In our example, there exists no direct assignment edge to $*y$, but our must-alias analysis determines that $*y$ and $*x$ must alias. As such, the direct assignment edge $z \xrightarrow{a} *x$ as well as the induced edges $z \overset{V}{\longleftrightarrow} *x$ and $\&c \overset{V}{\longleftrightarrow} *x$ are deleted. Details about strong update and edge deletion can be found in §3.5.



(a) g_1 **(b)** g_2 **(c)** Partition 1 after compression

Figure 2. Two frequent subgraphs mined over the PEGs in Partition P_1, with frequency ≥ 2 and size ≥ 3: (a) g_1 whose frequency = 4 and size = 7; (b) g_2 whose frequency = 2 and size = 4; (c) concise representation of P_1 based on g_1 and g_2.

Exploiting Locality between Consecutive PEGs. One clear advantage of our evolving graph formulation is that we can exploit similarities between PEGs for increased efficiency. In particular, Chianina extracts *frequent common subgraphs* (FCS) among PEGs and composes each PEG by *assembling existing FCSes* instead of duplicating these common edges and vertices in each PEG. In our example, P_1 consists of 4 PEGs. We invoke an off-the-shelf *itemset miner* Eclat [5] to discover the frequent edge-sets across these PEGs. Figure 2a and 2b depict two frequent subgraphs (g_1 and g_2), mined by using 2 as the frequency threshold and 3 as the size threshold. These two thresholds determine, respectively, the minimum occurrences of a subgraph and the minimum number of edges for the subgraph to be considered as a FCS. Next, Chianina *de-duplicates* PEGs by replacing each instance of g_1 and/or g_2 in each PEG with a reference. As shown in Figure 2c, $\mathcal{OUT}_3$ is now represented as a reference to g_1 and $\mathcal{OUT}_4$ as two references to g_1 and g_2. $\mathcal{OUT}_5$ and $\mathcal{OUT}_6$ are stored as a hybrid set of g_1 and g_2 references together with *residue edges* that do not belong to any FCS. Details of this algorithm is discussed in §3.4.

Dynamic Edge Pruning. Note that the pre-computed call graph may contain spurious calls due to the imprecision of the (inexpensive) points-to analysis used. To improve analysis precision, Chianina enables dynamic pruning of edges if our client is a pointer or alias analysis. Edge pruning can be easily done by checking the validity for edges connecting actual and formal parameters in the cloned control flow graph. The precise points-to set of the target variable computed by our system is used *on the fly* to determine whether such an edge is spurious. A spurious edge would not be traversed and hence everything reachable from it would not be traversed. A potential limitation is that it can be hard to pre-compute a proper call graph for certain dynamic languages such as JavaScript [19, 34, 43, 60]. To support such languages, future work can extend Chianina to explore call edges *on-the-fly* as part of the computation model.

Chianina is "Soundy". Like a typical static analysis [41], Chianina provides a sound solution if the program does not perform type casts between pointers and values of other types, and pointer arithmetic. Unsoundness can result from these language issues.

3 Chianina Design and Implementation

We architect Chianina as an *disk-based, out-of-core* graph system running on a single machine — since static analysis is our application domain, the desired system should run on developers' working machines, providing support for their daily development tasks. This section first discusses how a developer can use Chianina and then its design.

3.1 Programming Model

Similarly to the monotone framework [31, 33], implementing a client analysis on Chianina requires two tasks. First, the developer needs to create a subclass of an interface called `DataflowFactGraph` to specify her own graph implementation for dataflow facts. In the case of pointer/alias analysis, this subclass is PEG. Second, she implements two functions `combine` and `transfer`, which are used to merge dataflow facts at the control join points and propagate dataflow facts at statements, respectively.

As discussed earlier in §2.2, the frontend is a compiler pass that generates, by default, the CFG for each function, and each vertex of the CFG references another graph representing the dataflow fact at the vertex. The developer can also customize the format of the CFG. For our pointer analysis, we actually generates a more efficient sparse def-use graph proposed in [23].

Applicability. Chianina is a general framework supporting all context- and flow-sensitive analyses. In this paper, we implemented three particular analyses, pointer/alias analysis, null-value flow analysis, and cache-analysis as proof-of-concept examples. Flow-sensitive pointer/alias analysis serves as the foundation for virtually all static analyses. The null-value flow analysis is a representative of IFDS analyses (including value flow analysis, taint analysis, etc.) while cache analysis is an example of non-IFDS dataflow analysis.

Performance-wise, the heavier an analysis, the more benefit Chianina provides. For example, a fully context-sensitive analysis benefits the most because it can hardly be done on a commodity PC without out-of-core support. On the other hand, running an analysis that does not require much memory on Chianina may incur extra overheads.

3.2 Two-Level Parallel Computation

Parallel processing is key to our performance. It is enabled by cloning, which makes threading straightforward by physically separating CFGs under different contexts and eliminating most of the sharing between threads.

Algorithm 1 provides Chianina's iterative computation algorithm. Chianina exploits parallelism at two levels: (1) bulk *synchronous* parallel computation (BSP) at the partition level (Line 7) and (2) *asynchronous* computation at the CFG vertex level (Line 20). The loop between Line 5 and Line 16 describes a typical BSP style computation — partitions scheduled to process are loaded and processed completely in parallel during each superstep (*i.e.*, loop iteration). Each partition $\mathcal{P}_i$ has three data structures: (1) $\mathcal{F}_i$ — the active CFG vertices that form the frontier for the partition, (2) $\mathcal{G}_i$ — the set of dataflow fact graphs, and (3) $\mathcal{Q}_i$ — the message queue. In the beginning, $\mathcal{F}_i$ contains all vertices in the partition (Line 3).

The partition-level BSP computation is done by the loop from Line 7–10. Chianina loads the active vertices in $\mathcal{F}_i$ and the dataflow fact graphs $\mathcal{G}_i$ of each scheduled partition $\mathcal{P}_i$ into memory (Line 8), processes the partition (Line 9), and finds and exploits frequent common subgraphs (Line 10).

Function ProcessPartition describes the logic of processing of each partition that exploits parallelism at the (second) CFG-vertex level. Chianina iterates, in parallel, over the active CFG vertices in $\mathcal{F}_i$, applying the two user-defined functions Combine and Transfer on each vertex. The alias computation logic is done in Transfer. If the resulting PEG $Temp_k$ is *not isomorphic* to the previously computed $\mathcal{OUT}_k$ (Line 24), we record k into *changeset* and add k's CFG successors into the frontier set $\mathcal{F}_i$. It is clear that this parallel loop performs *asynchronous* computation — whenever a new active vertex is detected, it is added into $\mathcal{F}_i$ and *immediately* processed by a thread without any synchronization. Locks (omitted here) are used to guarantee data race freedom — no vertex will be processed simultaneously by multiple threads.

Asynchronous computation performs faster updates than synchronous computation at the cost of increased scheduling complexity. At the vertex level, since all CFG vertices of a partition are already in memory, asynchronous parallelism is a better fit as long as we can guarantee the data race freedom and atomicity of the transfer function execution for each vertex. However, at the partition level, our scheduler determines which partitions to load and run based on a set of already complex criteria, and hence, using BSP-style parallelism significantly simplifies our scheduler design.

Finally, the loop at Line 29 iterates over all CFG vertices whose dataflow facts have changed to find *mirror vertices* such as statement 4 in Figure 1a. In particular, we find the partition $\mathcal{P}_j$ that contains each mirror vertex s and puts

Algorithm 1: Two-level Parallel Computation.

1 $\mathcal{V} \leftarrow$ {all vertices in the cloned GCFG}
2 $\mathcal{G} \leftarrow$ {all initialized dataflow facts}
3 $[\mathcal{P}_0{:}\langle\mathcal{F}_0, \mathcal{G}_0\rangle, \ldots, \mathcal{P}_i{:}\langle\mathcal{F}_i, \mathcal{G}_i\rangle, \ldots] \leftarrow$ Partition($\mathcal{V}, \mathcal{G}$)
4 **repeat**
5   *scheduled* $\leftarrow$ Schedule()
6   /*Level 1: BSP computation at partition level*/
7   **for** *Partition* $\mathcal{P}_i \in$ *scheduled* **do in parallel**
8         $\langle\mathcal{F}_i, \mathcal{G}_i\rangle \leftarrow$ Load($\mathcal{P}_i$)
9   `ProcessPartition` ($\mathcal{F}_i, \mathcal{G}_i$)
10   CompressFCS($\mathcal{G}_i$)
11   **for** *Each partition* $\mathcal{P}_i$ **do in parallel**
12   **if** $\mathcal{Q}_i \neq \emptyset$ **then** $\mathcal{F}_i \leftarrow \mathcal{Q}_i$
13   **if** $\mathcal{P}_i \in$ *scheduled* **then** /*for loaded partitions*/
14   Write $\mathcal{G}_i, F_i$ back to disk
15   Delete $\mathcal{P}_i$ from memory
16 **until** $\forall i, \mathcal{F}_i = \emptyset$
17 **Procedure** `ProcessPartition`($\mathcal{F}_i, \mathcal{G}_i$)
18 *changeset* $\leftarrow \emptyset$
19 /*Level 2: Async. dataflow computation at stmt level*/
20 **for** *each CFG vertex* $k \in \mathcal{F}_i$ **do in parallel**
21   Remove k from $\mathcal{F}_i$
22     $\mathcal{IN}_k \leftarrow$ Combine(k)
23     $Temp_k \leftarrow$ Transfer($\mathcal{IN}_k$)
24   **if** $\neg$ *IsIsomorphic*($Temp_k, \mathcal{OUT}_k$) **then**
25         $\mathcal{OUT}_k \leftarrow Temp_k$
26   *changeset* $\leftarrow$ *changeset* $\cup \{k\}$
27         $\mathcal{F}_i \leftarrow \mathcal{F}_i \cup$ Successor(k) $\setminus Mirror$
28 /* Process CFG vertices with changed dataflow facts*/
29 **foreach** *CFG vertex* $k \in$ *changeset* **do**
30   **foreach** $s \in$ *Successor*(k) **do**
31   **if** *s is a mirror vertex* **then**
32             $\mathcal{Q}_j \leftarrow \{\langle s, \mathcal{OUT}_k\rangle\} \cup \mathcal{Q}_j$, where $s \in \mathcal{P}_j$

its dataflow fact graph $\mathcal{OUT}_k$ into its message queue $\mathcal{Q}_j$ (Line 32). Later, when all scheduled partitions are done with their processing (Line 11), the synchronization phase starts (Line 11 – Line 15), updating each partition $\mathcal{P}_i$'s active vertex set $\mathcal{F}_i$ with the messages in $\mathcal{Q}_i$ (received from the processing of other partitions). At the end of each superstep, the updated $\mathcal{G}_i$ and $\mathcal{F}_i$ are written back to disk and removed from memory (Line 13) if partition $\mathcal{P}_i$ is currently in memory.

3.3 Partitioning and Scheduling

Partitioning. Chianina uses the vertex-centric edge-cut strategy [44] for effective partitioning, which assigns CFG vertices to partitions and cuts certain edges across partitions. Specifically, vertices of the global control flow graph are firstly divided into disjoint sets. A partition is then created by assigning all the edges whose source or destination vertex belongs to this set. There often exist edges of the form $x \longrightarrow y$

that cross two partitions P_1 and P_2 (*e.g.*, $x \in P_1$ and $y \in P_2$). Chianina creates *mirror* vertices x' and y', and places the edges $x \rightarrow y'$ and $x' \rightarrow y$ into P_1 and P_2, respectively.

For each partition, its space is consumed by its CFG edges as well as dataflow fact graphs associated with its vertices (including mirror vertices). Dataflow fact graphs are maintained in a separate storage space from CFG edges. As a result of this partitioning scheme, for any vertex (except for mirrors) within a partition, Chianina can apply the transfer function on it by accessing and updating its incoming and outgoing dataflow facts. For each vertex whose successor is a mirror vertex, when its associated dataflow fact is updated, the mirror vertex is marked as active. A message containing the vertex ID and its updated dataflow fact graph is sent to its containing partition, as shown in Line 32 in Algorithm 1.

How to split GCFG nodes into disjoint sets determines the effectiveness of partitioning, which has further impact on the overall performance. Traditional graph partitioning schemes [8] minimize the number of cuts across partitions, with the goal to save communication costs. However, those schemes do not consider the unique characteristics of our (flow-sensitive analysis) workload. For example, the computation performed by a flow-sensitive analysis follows the structure of the CFG. It is well-known in the program analysis community that the convergence speed of an iterative analysis is significantly affected by the order in which CFG vertices are visited [13]. Intuitively, desirable performance can be achieved if all predecessors of a CFG vertex have been processed before the vertex itself, because the transfer function can just use the latest updates from its predecessors.

Based on the insight, we propose a *balanced, topology-based* partitioning mechanism. Given the number of partitions (specified by the user as a parameter) and the total number of vertices in the GCFG, we first calculate the average number of vertices for each partition. Next, the partitioner traverses the GCFG in a *topological order* (a.k.a. reverse post-order of DFS traversal), starting from each entry vertex of the GCFG. The traversal continues until the number of vertices visited matches (roughly) the average number. Once a partition is generated, we repeat the same process by using another unvisited vertex as the root. Eventually, all partitions are produced with balanced sets of vertices that also follow the traversal order.

This algorithm works well for CFGs without cycles. To deal with cycles (induced by loops), we compute strongly connected components (SCCs for brevity) over the GCFG. The nodes within a SCC are connected to each other. As a result, the control flow graph with cycles becomes an acyclic graph with SCCs. The above algorithm can then be conducted over the acyclic graph to produce balanced partitions.

Scheduling. Similarly to the partitioning scheme, the scheduler also needs to take into account topology when deciding which partitions to load and process. Due to dependencies induced by inter-partition edges (say $x \rightarrow y$), one major goal of the scheduler is to schedule the processing of the partition containing x before that of the partition containing y, so that communication costs can be reduced and the algorithm can converge quickly. To this end, we devise a *priority queue based* scheduling mechanism. We assign each partition a priority, which is a function of (1) the number of its active vertices (*i.e.*, the size of $\mathcal{F}_i$) and (2) whether or not the partition is currently in memory. The more active vertices a partition has, the more updates can be generated during computation. Furthermore, if a partition is already in memory, processing it again in the next superstep can save the large cost of a memory-disk round trip.

Our scheduler selects a number N of partitions with the highest priority. The value of N is determined by (1) the amount of memory each partition is estimated to consume, (2) the total amount of available memory, and (3) the number of CPU cores. Our goal is to fully utilize the memory and CPU resources without creating extra stress.

3.4 FCS-Based De-Duplication

Although Chianina divides the input into many small partitions, partitions are still space-consuming especially because each CFG vertex carries a dataflow fact graph. These graphs exhibit both *temporal* and *spatial* locality — graphs belonging to connected CFG vertices are processed contiguously and have large overlap. To exploit such overlaps, we propose a frequent-itemset-based approach to find frequent common subgraphs and perform de-duplication by maintaining only one instance for each FCS and replacing other instances with references. De-duplication (Line 10 in Algorithm 1) is conducted before writing dataflow facts back to disk. In particular, our algorithm models each dataflow fact graph (*e.g.*, PEG) as an itemset where each item is an edge. The graph miner discovers frequent itemsets, each of which occurs at least N times (*i.e.*, N is a threshold) among dataflow fact graphs in the same partition.

Once these FCSes are mined, we check each dataflow fact graph and see if it contains any FCSes. If it does, we replace each instance of each FCS with a reference, as illustrated in Figure 2c. Given multiple FCSes, there may exist multiple ways to conduct the placement. Given that the benefits of de-duplication are determined primarily by an FCSes' frequency and size. The higher these numbers are, the more benefit can be reaped. As such, Chianina assigns each FCS mined a priority score, computed as the product of its frequency and size. A greedy algorithm is then used to apply candidate FCSes in the descending order of their priority.

In Chianina, we leverage an off-the-shelf frequent itemset mining tool Eclat [5] to uncover FCSes. Although leveraging these FCSes significantly reduces the size of dataflow facts, it inevitably introduces overhead. With the growth in both the number and size of dataflow facts, the mining cost is non-trivial — it can take several minutes to run each mining task for large partitions in our experiments. To reduce the overhead, we can focus only on very frequent and/or very large FCSEs by raising the mining thresholds. Moreover, we randomly sample the dataflow fact graphs in each partition, selecting no more than 10K graphs as our mining dataset. These two approaches collectively bring the overhead down to an acceptable percentage (*i.e.*, less than 5%).

3.5 Strong Update and Edge Deletion

As stated earlier, the dataflow transfer function `transfer` needs to be provided by the developer. For pointer/alias analysis, the `transfer` function not only applies the logic of *GEN* and *KILL*, but also discovers transitive edges on each PEG to compute an alias solution. The logic of *GEN* is straightforward — Rule 1–4 in §2.1 clearly describes how new edges should be added. The algorithm of computing the alias solution from a PEG is based on CFL-reachability [35, 59] (shown in Equation 5 and 6) and well-known to the community [80]. Hence, we do not include this algorithm in the paper. The logic of *KILL* (*i.e.*, edge deletion) involves strong update, which is crucial for achieving high precision of flow-sensitive analysis [14, 37, 62]. Since this logic is much trickier than that for edge addition, here we focus on the discussion of edge deletion.

Condition for Strong Update. Strong update can be enabled on pointer expression x such that x is guaranteed to refer to a single memory location (*i.e.*, *singleton*) throughout the execution. We follow [37] to identify our singleton set. The detailed algorithm is known and omitted from the paper to save space. Informally, a local or global variable is singleton except for the following cases: (1) dynamically allocated variables, where one abstract variable may correspond to multiple memory locations during execution; (2) local variables of recursive procedures (either directly or transitively recursive), where each variable may have multiple instances on the stack; and (3) array variables where usually only one element is updated.

Edges to Delete. When such an expression (*e.g.*, $*p = v$) is defined, strong update may be performed because the value contained in the location l pointed-to by p changes. This removes the value-aliasing (Equation 5) between $*p$ and any pointer variables that previously receive their values from the location. On the PEG, two kinds of edges need be deleted: all (direct and transitive) edges (a) *going into* and (b) *coming out of* any pointer expressions referring to l. For (a), there are four sub-cases: (a.1) direct assignment edges going to expression $*p$, added due to a previous statement such as $*p = x$ — such a relationship no longer holds; (a.2) direct assignment edges going to expression $*q$ such that p and q *must alias*. p and q must alias if they both have only one and the same memory location o in their points-to set and o is a singleton memory location; (a.3) transitive (V- or M-) edges going to expression $*p$ — these edges represent aliasing relationships between the old value inside $*p$ and another pointer expression and thus need to be deleted; and (a.4) transitive (V- or M-) edges going to expression $*q$ such that p and q *must alias*; these edges need to be deleted for similar reasons. We need to remove not only edges *going into* $*p$, but also edges *coming out* of $*p$. For example, a direct edge coming out of $*p$ due to a previous statement $v = *p$ needs to be deleted, since v is not longer related to $*p$ which now contains a different value. Similarly to (a), four sub-cases exist in (b), which need to be deleted as well.

Table 1. Characteristics of subject programs.

Subject	Version	#LoC	#Inlines	#V-CFG	#E-CFG	Description
Linux	5.2	17.5M	48.5M	443.5M	668.7M	OS
Firefox	67.0	7.9M	22.2M	283.5M	504.9M	web browser
PostgreSQL	12.2	1.0M	5.4M	39.3M	80.4M	database
OpenSSL	1.1.1	519K	4.5M	49.4M	99.3M	protocol
Httpd	2.4.39	196K	293K	2.6M	3.8M	web server

4 Evaluation

Our evaluation focuses on the following three questions:

- Q1: How does Chianina perform? How does it compare to other analysis implementations? (§4.1)
- Q2: How effective are our de-duplication, partitioning, and scheduling? (§4.2)
- Q3: Is the extra precision gained from context- and flow-sensitivity useful in practice? (§4.3)

We selected five large software systems including the Linux kernel, Firefox, PostgreSQL, OpenSSL, and Apache Httpd as our analysis subjects. We implemented three context- and flow-sensitive analyses on top of Chianina: a pointer/alias analysis discussed in the paper as an example, a null-value flow analysis with context-sensitive heap tracking, as well as an instruction cache analysis with 512 cache lines and LRU replacement policy. The null-value analysis was conducted in parallel with the pointer/alias analysis — because pointer information is needed to track flows into/out of the heap, this analysis implements its dataflow fact graph by augmenting the PEG representation from the pointer/alias analysis with additional types of vertices representing null or non-null values. For the cache analysis, we adopted the same abstract cache model as [71], which represents a program as a set of instructions and their associated ages. The analysis computes a cache model at each program point and determines whether the instruction at the point leads to a cache hit or miss.

The Chianina-based implementation for the pointer/alias analysis has **553** lines of C++ code, most of which are on the implementation of CFL-reachability and strong update. In contrast, a context-, flow-insensitive pointer analysis [37] (that supports strong update) has **2499** lines of C++ code, while the staged context-insensitive, flow-sensitive analysis for C [24] has **10,649** lines. The implementations for other two analyses (null-value flow and cache analysis) have **708** and **436** lines of C++ code, respectively.

Table 2. Chianina performance: columns shown are numbers of partitions (#Part.), numbers of iterations needed to converge (#Ite.), total numbers of vertices (#V-PEGs) and edges (#E-PEGs) in the GCFG for alias and null-value flow analysis, total numbers of cache states (#States) for cache analysis, and analysis times (Time), respectively.

	Alias analysis					NULL value flow analysis with alias tracking					Cache analysis			
Subject	#Part.	#Ite.	#V-PEGs	#E-PEGs	Time	#Part.	#Ite.	#V-PEGs	#E-PEGs	Time	#Part.	#Ite.	#States	Time
Linux	287	339	5.9B	126.1B	20.9hrs	290	355	6.1B	126.2B	22.6hrs	232	4364	18.9B	24.4hrs
Firefox	150	183	3.4B	84.2B	11.4hrs	150	193	3.8B	85.0B	12.5hrs	158	1949	9.6B	10.6hrs
PostgreSQL	34	43	482.1M	13.7B	1.3hrs	42	45	513.6M	13.7B	1.5hrs	30	808	1.3B	2.4hrs
OpenSSL	12	21	442.1M	5.7B	55.3mins	12	22	468.1M	5.7B	59.8mins	31	582	1.1B	2.7hrs
Httpd	1	1	37.6M	585.4M	4.7mins	1	1	41.2M	589.3M	5.0mins	2	17	110.2M	7.3mins

As discussed earlier, context sensitivity is achieved by aggressive function cloning. Table 1 reports the static characteristics of each subject including its version information, the number of lines of code excluding whitespace and comments (#LoC), the number of functions inlined (#Inlines), the numbers of CFG vertices (#V-CFG) and edges (#E-CFG) in the global CFG after cloning, and the type description.

All the experiments were conducted on a *commodity desktop* with an Intel Xeon W-2145 8-Core CPU, 16GB memory, and 1T SSD, running Ubuntu 16.04. This resource profile is consistent with that of developers' working machines.

4.1 Chianina Performance

Table 2 reports, for the three client analyses, a variety of performance statistics including numbers of partitions generated, numbers of iterations (supersteps) needed for convergence, total numbers of vertices and edges in all PEGs for alias and null value flow analysis, total number of cache states (*i.e.*, pair of instruction and its age) in the cache models for cache analysis, and total computation times.

The numbers of partitions for large programs such as Linux and Firefox are greater than 100. It would not have been possible to scale the analysis to such large programs without our disk support. Overall, it took the three analyses 20.9, 22.6, and 24.4 hours to process the entire Linux kernel in a context- and flow-sensitive fashion. These analyses converged much faster for smaller programs such as Httpd (in a few minutes), which can be analyzed as only one or two partitions.

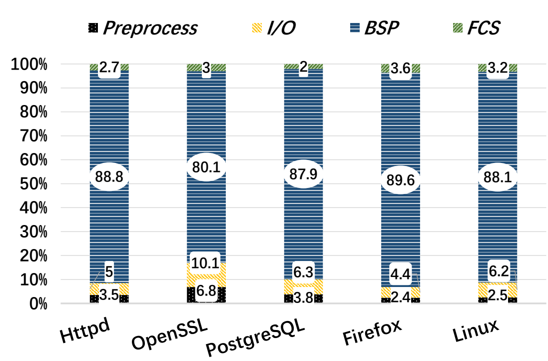


Figure 3. Performance breakdown of alias analysis: for each subject, shown *bottom-up* are fractions of preprocessing, I/O, BSP computation, and FCS de-duplication.

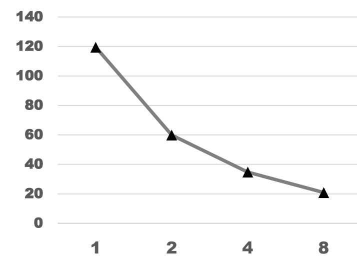


Figure 4. Alias analysis on Linux: time (in hours) with varying numbers of threads.

Performance Breakdown. To better understand the performance, we further broke down the alias analysis execution into four phases – preprocessing (*i.e.*, partitioning), disk I/O (*i.e.*, reading/writing partitions), (in-memory) BSP computation, and FCS de-duplication – and measured the time spent on each phase. Figure 3 depicts the time breakdown. As shown, the in-memory BSP computation dominates the execution. For example, it takes around or more than 80% of the time for all five programs, indicating that these analyses are compute-intensive. This is expected because each iteration updates hundreds of millions of PEGs, each of which can have thousands of edges. This observation suggests that more CPU resources (*e.g.*, cores, GPUs, or cluster) should be enlisted to further improve performance. Time spent on I/O varies across programs; for Linux, it takes around 6% of the total execution time. This fraction is reasonably small due to use of modern SSDs that have much higher bandwidth and lower read/write latency than HDDs. The cost of FCS de-duplication is constantly lower than 4%, thanks to the optimizations discussed in §3.4.

Parallel Scalability. To understand Chianina's (thread) scalability, we measured the alias analysis time on Linux for varying numbers of threads used in the system. As shown in Figure 4, Chianina scales almost linearly with the number of threads because cloning eliminates most of the data sharing between threads. In contrast, most existing analyses are single-threaded. Even for multi-threaded implementations [3, 53], it is hard for them to achieve such a speedup without physical separation of functions under different contexts (enabled by cloning).

Existing Analyses. The goal of this comparison is to understand if our context- and flow-sensitive alias analysis is

Table 3. Performance comparison for context-insensitive, flow-sensitive pointer analysis; OOM indicates out-of-memory; - indicates runtime error.

	Linux	Firefox	PostgreSQL	OpenSSL	Httpd
Reference[24]	OOM	OOM	14.7mins	OOM	34.7s
SVF[63]	-	OOM	56.1s	OOM	8.3s
Chianina	1.9hrs	4.2hrs	3.9mins	25.7mins	11.5s

more scalable and efficient than state-of-the-art analysis implementations. However, we could not find any available implementation of the same analysis for C. Yu *et al.* [77] and Kahlon [29] reported the context- and flow-sensitive pointer analyses, but neither of the implementations is publicly available. Hardekopf *et al.* [23, 24] and Lhotak and Chung [37] have both implemented the variations of flow-sensitive but context-insensitive pointer analysis for C. Although their implementations are available online, they were developed a long time ago for deprecated versions of LLVM, which are incompatible with the subject programs and our OS. Doop [6] is a context-sensitive analysis framework, but it only supports Java and does not have a C frontend. The only available tool we can run is SVF [63], a demand-driven flow-sensitive analysis tool, which does not support whole-program context-sensitive pointer analysis.

Since no existing implementation for both context- and flow-sensitive pointer/alias analysis was available for direct comparison, we implemented by ourselves the staged flow-sensitive pointer/alias analysis, by faithfully following the algorithm described in [24]. The original analysis in [24] does not consider context sensitivity, and hence, we added context sensitivity to our implementation by letting the analysis take as input the cloned GCFG, which is automatically context sensitive (we cannot do this to SVF due to different implementation bases). We compared Chianina with this version in a fully context-sensitive, flow-sensitive manner. This reference implementation failed to analyze most programs except for Httpd in our benchmark set — it ran out of memory quickly in a few seconds. This is not surprising as holding the GCFG for large programs requires a huge amount of memory. For Httpd, the reference implementation (single-threaded) takes more than 20 minutes and is much slower than Chianina. This is due to the high parallelism degree in Chianina, which is, in turn, enabled by cloning and our out-of-core support.

Next, we disabled context sensitivity in Chianina, enabling *direct* comparisons between Chianina, SVF, and the reference implementation of [24]. In this setting, all the three tools ran context-insensitive, flow-sensitive pointer analysis. Table 3 reports the analysis times the three tools took to analyze the five programs. Without context sensitivity, the reference implementation still failed to analyze Linux, FireFox and OpenSSL due to out-of-memory errors. SVF ran out of memory for Firefox and OpenSSL, and crashed on Linux. For Httpd and PostgreSQL, all the tools successfully analyzed them. Chianina outperformed [24] thanks to parallel processing. For PostgreSQL, however, SVF achieved better performance than Chianina. This is easy to understand — many optimizations Chianina performs for scalability purposes (*e.g.*, preprocessing, scheduling, disk I/O, and FCS de-duplication) take time to run; if scalability is not a concern, these optimizations would only add overhead.

Precision and Correctness Validation. We first compared the precision of flow-sensitivity among the three analyses in Table 3 (Chianina is in its context-insensitive version) using the *alias-set metric*. Particularly, we examined each pointer dereference expression in *load* and *store* statements of the program, and measured *the average sizes of their alias sets* weighted by the number of times each variable is dereferenced — the smaller the better. On Httpd and PostgreSQL, for which these three flow-sensitive analyses scale, they achieve almost the same average sizes, with a less than 0.5% variation, indirectly validating the correctness of our implementation.

We further verified Chianina's correctness by testing it over a micro-benchmark set PTABen [1] in both context- and flow-sensitive settings. Our analysis passed all assertions.

4.2 De-Duplication, Partitioning and Scheduling

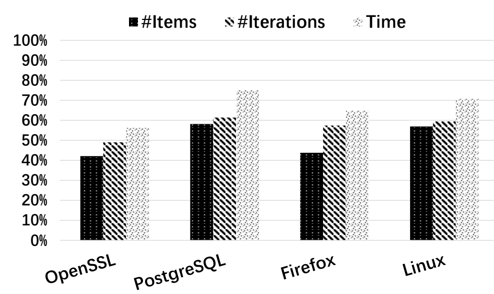


Figure 5. Percentages in numbers of PEG edges, numbers of iterations (*i.e.*, supersteps) needed, and total time spent for Chianina + FCS, using Chianina - FCS as the baseline (100%).

To understand the performance impact of de-duplication, we compared two versions of Chianina, one with FCS de-duplication enabled (Chianina + FCS) and another without (Chianina - FCS). We ran these two versions under the same configuration and inputs for alias analysis, and collected the relevant execution statistics. Figure 5 depicts the numbers of PEG edges, numbers of iterations needed for convergence, and total time spent for Chianina + FCS, as a fraction of those of Chianina - FCS (*i.e.*, the baseline). Note that since Httpd is a small program with only one single partition, we excluded it from the set for the FCS evaluation. As shown, de-duplication significantly improved all of these aspects. For example, the overall time is reduced by more than 30% on average when FCS de-duplication is enabled.

To understand the efficacy of our partitioning and scheduling, we collected the statistics for alias analysis in a similar manner by running two versions of Chianina, one with our

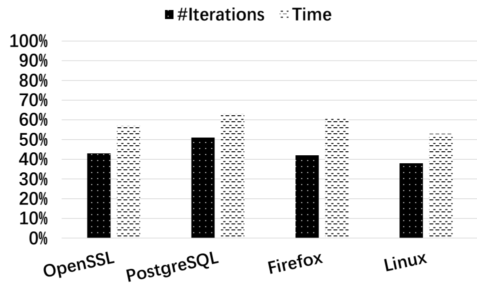


Figure 6. Percentages in numbers of iterations and total time for Chianina + PS using Chianina - PS as baseline (100%); Httpd is not considered here as it has one single partition.

partitioning and scheduling algorithm (Chianina + PS) and a second that uses default algorithms (Chianina - PS) — in particular, in the second version, we partitioned the GCFG using the min-cut algorithm [44] and scheduled random partitions (with active vertices) for processing in each superstep. We use Chianina - PS as the baseline and report the statistics for Chianina + PS as a fraction in relation to the baseline in Figure 6. The statistics considered include numbers of iterations needed for convergence, and time spent. As shown, our partitioning and scheduling algorithms are effective — they significantly improve the efficiency in all these aspects. For example, total running time is reduced by more than 40% by employing our structure-aware partitioning and scheduling.

Table 4. Sizes of alias sets of pointer expressions involved in *load* and *store* statements under three different pointer/alias analyses – our context-sensitive and flow-sensitive (CF), context-insensitive and flow-sensitive (F), context-sensitive and flow-insensitive (C).

	Load			Store		
Subject	CF	F	C	CF	F	C
Linux	0.24	0.54	7.18	0.31	0.95	5.54
Firefox	0.29	0.70	5.08	0.14	1.51	3.80
PostgreSQL	0.44	1.54	14.0	1.11	1.57	18.1
OpenSSL	0.77	4.06	10.38	0.08	0.25	0.61
Httpd	0.38	1.46	11.72	1.97	1.97	10.80

4.3 Usefulness of Gained Precision

To understand the gained accuracy of our context- and flow-sensitive alias analysis, we used the same alias-set metric to compare precision among three variants of Chianina– the full context- and flow-sensitive analysis (CF), a context-insensitive, flow-sensitive analysis (F), and a context-sensitive, flow-insensitive analysis (C). Table 4 reports the average sizes of alias sets for each analysis. Clearly, our flow- and context-sensitive analysis has the highest precision. The context-sensitive and flow-insensitive analysis (C) has the largest number (*i.e.*, lowest precision). This observation demonstrates that flow-sensitivity is more important than context-sensitivity for large C programs because analysis precision loses significantly if strong update is disabled.

Table 5. Checkers implemented including the dataflow analysis-based null pointer dereference (DF-Null), use-after-free (DF-UAF), double free (DF-DF) and the belief analysis-based null pointer dereference (BA-Null), their numbers of bugs reported by the baseline checkers augmented with our context- and flow-insensitive analysis (base+CF), context-sensitive and flow-insensitive (base+C), context-insensitive and flow-sensitive (base+F) in the Linux kernel 5.2.

Checker	DF-Null	DF-UAF	DF-DF	BA-Null	total
base+CF	196	647	193	620	1656
base+C	217	1144	212	723	2296
base+F	211	805	200	663	1879

To measure the real-world usefulness of the increased precision, we implemented four static checkers: (1) a dataflow-based null pointer dereference checker, (2) a use-after-free checker, (3) a double-free checker, and (4) a belief analysis based null pointer dereference checker. The first three checkers were commonly used in the program analysis community [52, 67] and the last checker was used in the classical bug study done by Engler *et al.* [7, 18]. Note that the original versions of these checkers do not use any pointer information; they only use heuristics. To understand the effectiveness of our flow-sensitive alias analysis, we augmented these checkers with alias information provided by three analyses — our *context-* and *flow-sensitive* analysis (CF), context-sensitive, flow-insensitive analysis (C), and context-insensitive and flow-sensitive analysis (F). We next compared the numbers of warnings generated by these four checkers when augmented with each of these three pieces of alias information. The fewer warnings generates, the better (*i.e.*, more false positives are pruned). Table 5 reports these numbers — a large number of false warnings are pruned by enabling context and flow sensitivity. Similarly to an earlier observation, flow sensitivity seems more important than context sensitivity as well in pruning false warnings.

5 Related Work

Evolving Graph Systems. Although we formulate flow-sensitive analysis as an evolving graph processing problem, the nature of the problem differs significantly from that dealt with in the graph system community [22, 27, 45, 65]. System design depends on (1) data and (2) computation. On the data side, each vertex of the graph in Chianina is associated with a separate dataflow graph. This kind of graphs differs significantly from the typical evolving graphs where no semantic dependence exists between vertices. On the computation side, the computation in Chianina is defined by vertex types — each vertex (statement) performs arbitrary edge addition/deletion based on the statement's semantics and client type. This computation model differs from the computation in existing systems, which is driven solely by the graph algorithm (*e.g.*, PageRank) and has nothing to do with the graph itself. In summary, the semantics of program analysis makes Chianina distinctive and none of existing systems are able to perform this type of computation.

Flow-Sensitive Analyses. A common optimization of scaling flow-sensitive analysis is to perform a sparse analysis preventing redundant values from being propagated [10, 51]. Hind and Pioli [26] adopted the sparse evaluation graph [12] which eliminates pointer-free statements from the CFG. Hardekopf and Lin [23, 24] proposed to utilize a semi-sparse representation by connecting variable definitions with their uses, allowing dataflow facts to be propagated only to the locations needing the variable. Sui and Xue implemented SVF [63], which constructs the sparse value-flow graph and performs the pointer analysis in an iterative manner. Other techniques such as [25, 64] use similar ideas to scale flow-sensitive analysis. To accelerate an interprocedural dataflow analysis, a few techniques attempt to parallelize its computation. Rodriguez *et al.* [53] proposed an actor-model-based parallel algorithm for IFDS problems. Garbervetsky *et al.* [20] developed a distributed worklist algorithm using the actor model to implement a call-graph analysis. Albarghouthi *et al.* [3] parallelize a top-down interprocedural analysis using a MapReduce-like computation model. Several studies [49, 79] attempt to parallelize flow-sensitive pointer analysis. Since they all require large amounts of memory, there is no evidence that these approaches can scale to the Linux kernel.

Context-Sensitive Analyses. Generally, there are two dominant approaches to context-sensitive interprocedural analysis: the *summary-based approach* and the *cloning-based* approach [56]. The summary-based approach [11, 47, 52, 55, 70, 72, 76] constructs a summary (transfer) function for each procedure, and directly applies the summary to the specific inputs at the call site invoking the function. Although it is scalable for certain cases, it does not provide complete alias information for each particular context due to lack of explicit representation of calling contexts. Furthermore, it is difficult for certain analyses (*e.g.*, pointer analysis) to establish a succinct summary for each function and precisely model heap effects. The cloning-based approach [17, 69, 73, 74] provides complete information. However, it requires each procedure to be re-analyzed under each calling context and hence is hard to scale. Demand-driven techniques [9, 59, 75] match call/return edges *on the fly* for context sensitivity. A body of techniques have also been proposed to perform selective context sensitivity [32, 39, 40, 42, 46, 50, 57, 58, 78], so as to find sweatspots between scalability and precision.

Systems for Static Analyses. BDDBDDB [69] and Doop [6] are the early pioneers that run sophisticated static analysis on Datalog engines. These Datalog engines (even including a recent one *Soufflé* [28]) 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- and flow-sensitive analysis to large-scale systems like Linux on the commodity desktop we used. Weiss et al. [68] presents a database-backed static analysis for error propagation. A recent piece of work Graspan [67] aims to scale context-free language (CFL) reachability based analyses to large programs with disk support. Although Chianina is inspired by the same high-level observation as Graspan, it is impossible to extend Graspan to support arbitrary flow-sensitive analyses without re-designing the system from scratch. The simple computation logic for graph reachability does not work for Chianina's complex dataflow semantics. As an extension to Graspan, BigSpa [21, 81] adapts the same computation model to a distributed setting. Grapple [82] supports path sensitivity by concisely encoding path constraints. However, neither of them process evolving graphs or support flow-sensitive analyses that we focus on in this paper. Google [54] and Facebook [16] also deployed their analysis tools in the parallel/distributed setting to analyze their large-scale codebases. Chianina is another quest in this direction scaling context- and flow-sensitive analyses to large programs while requiring developers to provide only basic analysis algorithms.

6 Conclusion

This paper presents Chianina, a novel evolving graph system for scalable context- and flow-sensitive analysis for C code. Chianina requires developers to provide only the basic algorithm while leveraging system-level optimizations for scalability and efficiency. Using Chianina, a fully context- and flow-sensitive pointer/alias analysis can scale to modern large codebase like Linux kernel.

Future work can extend Chianina to analyze other languages as well. Chianina currently needs a pre-computed call graph to perform cloning. It can be hard to pre-compute a proper call graph for certain dynamic languages such as JavaScript. One potential extension is to add support for constructing the call graph *on the fly* based on pointer information computed. Moreover, adapting our work to a cloud setting is also a worthy task so as to further boost analysis scalability. The architecture of Chianina involving parallel processing model, partitioning and scheduling, is immediately applicable to the cluster/cloud settings.

Acknowledgments

We thank the anonymous reviewers and especially our shepherd Sandeep Dasgupta for their valuable comments and feedback. Our thanks also go to Yulei Sui for his feedback on SVF. This work was partially supported by the National Natural Science Foundation of China (No. 62032010, 61802168, and 62002157), the Natural Science Foundation of Jiangsu Province (No. BK20191247), the Fundamental Research Funds for the Central Universities (No. 14380065), the US National Science Foundation under grants CNS-1613023, CNS-1703598, CNS-1763172, CNS-2006437, and CNS-2007737, and the US Office of Naval Research under grants N00014-16-1-2913 and N00014-18-1-2037.

References

[1] [n.d.]. PTABen: a micro-benchmark suite for pointer analysis. https://github.com/SVF-tools/Test-Suite. Accessed: 2020-11-17.

[2] Alex Aiken, Suhabe Bugrara, Isil Dillig, Thomas Dillig, Brian Hackett, and Peter Hawkins. 2007. An Overview of the Saturn Project. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (San Diego, California, USA) *(PASTE '07)*. 43–48. https://doi.org/10.1145/1251535.1251543

[3] Aws Albarghouthi, Rahul Kumar, Aditya V. Nori, and Sriram K. Rajamani. 2012. Parallelizing Top-down Interprocedural Analyses. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. 217–228. https://doi.org/10.1145/2254064.2254091

[4] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. [n.d.]. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. 259–269. https://doi.org/10.1145/2594291.2594299

[5] Christian Borgelt. 2017. Find Frequent Item Sets with the Eclat Algorithm. http://www.borgelt.net/doc/eclat/eclat.html.

[6] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. 243–262. https://doi.org/10.1145/1640089.1640108

[7] Fraser Brown, Andres Nötzli, and Dawson Engler. 2016. How to Build Static Checking Systems Using Orders of Magnitude Less Code. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. 143–157. https://doi.org/10.1145/2872362.2872364

[8] Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. 2016. *Recent Advances in Graph Partitioning*. Springer International Publishing, Cham, 117–158.

[9] 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 *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. 196–210. https://doi.org/10.1145/3192366.3192378

[10] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. 1990. Analysis of Pointers and Structures. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation* (White Plains, New York, USA) *(PLDI'90)*. 296–310. https://doi.org/10.1145/93542.93585

[11] Ben-Chung Cheng and Wen-Mei W. Hwu. 2000. Modular Interprocedural Pointer Analysis Using Access Paths: Design, Implementation, and Evaluation. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. 57–69. https://doi.org/10.1145/349299.349311

[12] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. 1991. Automatic Construction of Sparse Data Flow Evaluation Graphs. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Orlando, Florida, USA) *(POPL '91)*. 55–66. https://doi.org/10.1145/99583.99594

[13] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. 2004. *Iterative data-flow analysis, revisited*. Technical Report.

[14] Arnab De and Deepak D'Souza. 2012. Scalable Flow-Sensitive Pointer Analysis for Java with Strong Updates. In *Proceedings of the 26th European Conference on Object-Oriented Programming* (Beijing, China) *(ECOOP'12)*. 665–687. https://doi.org/10.1007/978-3-642-31057-7_29

[15] Isil Dillig, Thomas Dillig, and Alex Aiken. 2008. Sound, Complete and Scalable Path-Sensitive Analysis. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. 270–280. https://doi.org/10.1145/1375581.1375615

[16] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling Static Analyses at Facebook. *Commun. ACM* 62, 8 (July 2019), 62–70. https://doi.org/10.1145/3338112

[17] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. 1994. Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation* (Orlando, Florida, USA) *(PLDI '94)*. 242–256. https://doi.org/10.1145/178243.178264

[18] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. 2000. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *Proceedings of the 4th Conference on Symposium on Operating System Design and Implementation - Volume 4* (San Diego, California) *(OSDI'00)*. Article 1.

[19] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Efficient Construction of Approximate Call Graphs for JavaScript IDE Services. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. 752–761.

[20] Diego Garbervetsky, Edgardo Zoppi, and Benjamin Livshits. 2017. Toward Full Elasticity in Distributed Static Analysis: The Case of Callgraph Analysis. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. 442–453. https://doi.org/10.1145/3106237.3106261

[21] 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 (April 2021), 867–883. https://doi.org/10.1109/TPDS.2020.3036190

[22] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. 2014. Chronos: A Graph Engine for Temporal Graph Analysis. In *Proceedings of the Ninth European Conference on Computer Systems* (Amsterdam, The Netherlands) *(EuroSys '14)*. Article 1, 14 pages. https://doi.org/10.1145/2592798.2592799

[23] Ben Hardekopf and Calvin Lin. 2009. Semi-Sparse Flow-Sensitive Pointer Analysis. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. 226–238. https://doi.org/10.1145/1480881.1480911

[24] Ben Hardekopf and Calvin Lin. 2011. Flow-Sensitive Pointer Analysis for Millions of Lines of Code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. 289–298.

[25] Rebecca Hasti and Susan Horwitz. 1998. Using Static Single Assignment Form to Improve Flow-Insensitive Pointer Analysis. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (Montreal, Quebec, Canada) *(PLDI'98)*. 97–105. https://doi.org/10.1145/277650.277668

[26] Michael Hind and Anthony Pioli. 1998. Assessing the Effects of Flow-Sensitivity on Pointer Alias Analyses. In *Proceedings of the 5th International Symposium on Static Analysis (SAS'98)*. 57–81.

[27] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. 2016. Time-evolving Graph Processing at Scale. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems* (Redwood Shores, California) *(GRADES '16)*. Article 5, 6 pages.

[28] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). 422–430.

[29] Vineet Kahlon. 2008. Bootstrapping: A Technique for Scalable Flow and Context-Sensitive Pointer Alias Analysis. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. 249–259. https://doi.org/10.1145/1375581.1375613

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

[31] John B. Kam and Jeffrey D. Ullman. 1977. Monotone Data Flow Analysis Frameworks. *Acta Inf.* 7, 3 (Sept. 1977), 305–317.

[32] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid Context-Sensitivity for Points-to Analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. 423–434. https://doi.org/10.1145/2491956.2462191

[33] Gary A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Boston, Massachusetts) *(POPL '73)*. 194–206. https://doi.org/10.1145/512927.512945

[34] Yoonseok Ko, Hongki Lee, Julian Dolby, and Sukyoung Ryu. 2015. Practically Tunable Static Analysis Framework for Large-Scale JavaScript Applications. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering* (Lincoln, Nebraska) *(ASE '15)*. 541–551. https://doi.org/10.1109/ASE.2015.28

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

[36] Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical for the Real World. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. 278–289. https://doi.org/10.1145/1250734.1250766

[37] Ondrej Lhoták and Kwok-Chiang Andrew Chung. 2011. Points-to Analysis with Efficient Strong Updates. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. 3–16. https://doi.org/10.1145/1926385.1926389

[38] Ondřej Lhoták and Laurie Hendren. 2006. Context-Sensitive Points-to Analysis: Is It Worth It?. In *Proceedings of the 15th International Conference on Compiler Construction* (Vienna, Austria) *(CC'06)*. 47–64. https://doi.org/10.1007/11688839_5

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

[40] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Scalability-First Pointer Analysis with Self-Tuning Context-Sensitivity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. 129–140. https://doi.org/10.1145/3236024.3236041

[41] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In Defense of Soundiness: A Manifesto. *Commun. ACM* 58, 2 (Jan. 2015), 44–46. https://doi.org/10.1145/2644805

[42] Jingbo Lu and Jingling Xue. 2019. Precision-Preserving yet Fast Object-Sensitive Pointer Analysis with Partial Context Sensitivity. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 148 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360574

[43] Magnus Madsen, Benjamin Livshits, and Michael Fanning. 2013. Practical Static Analysis of JavaScript Applications in the Presence of Frameworks and Libraries. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. 499–509. https://doi.org/10.1145/2491411.2491417

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

[45] Youshan Miao, Wentao Han, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Enhong Chen, and Wenguang Chen. 2015. ImmortalGraph: A System for Storage and Analysis of Temporal Graphs. *ACM Trans. Storage* 11, 3, Article 14 (July 2015), 34 pages. https://doi.org/10.1145/2700302

[46] 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. https://doi.org/10.1145/1044834.1044835

[47] Brian R. Murphy and Monica S. Lam. 1999. Program Analysis with Partial Transfer Functions. In *Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation* (Boston, Massachusetts, USA) *(PEPM '00)*. 94–103. https://doi.org/10.1145/328690.328703

[48] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. 439–455. https://doi.org/10.1145/2517349.2522738

[49] Vaivaswatha Nagaraj and R. Govindarajan. 2013. Parallel Flow-Sensitive Pointer Analysis by Graph-Rewriting. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques* (Edinburgh, Scotland, UK) *(PACT '13)*. 19–28.

[50] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. 2014. Selective Context-Sensitivity Guided by Impact Pre-Analysis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. 475–484. https://doi.org/10.1145/2594291.2594318

[51] John H. Reif and Harry R. Lewis. 1977. Symbolic Evaluation and the Global Value Graph. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California) *(POPL '77)*. 104–118. https://doi.org/10.1145/512950.512961

[52] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. 49–61. https://doi.org/10.1145/199448.199462

[53] Jonathan Rodriguez and Ondřej Lhoták. 2011. Actor-Based Parallel Dataflow Analysis. In *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software* (Saarbrücken, Germany) *(CC'11/ETAPS'11)*. 179–197.

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

[55] Mooly Sagiv, Thomas Reps, and Susan Horwitz. 1996. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. *Theor. Comput. Sci.* 167, 1-2 (Oct. 1996), 131–170.

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

[57] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: Fast and Precise Sparse Value Flow

Analysis for Million Lines of Code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. 693–706. https://doi.org/10.1145/3192366.3192418

[58] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-Sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. 17–30. https://doi.org/10.1145/1926385.1926390

[59] Manu Sridharan and Rastislav Bodík. 2006. Refinement-based Context-sensitive Points-to Analysis for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. 387–400. https://doi.org/10.1145/1133981.1134027

[60] Benno Stein, Benjamin Barslev Nielsen, Bor-Yuh Evan Chang, and Anders Møller. 2019. Static Analysis with Demand-Driven Value Refinement. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 140 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360566

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

[62] Yulei Sui and Jingling Xue. 2016. On-Demand Strong Update Analysis via Value-Flow Refinement. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. 460–473. https://doi.org/10.1145/2950290.2950296

[63] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural Static Value-Flow Analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC 2016)*. 265–266. https://doi.org/10.1145/2892208.2892235

[64] Teck Bok Tok, Samuel Z. Guyer, and Calvin Lin. 2006. Efficient Flow-Sensitive Interprocedural Data-Flow Analysis in the Presence of Pointers. In *Proceedings of the 15th International Conference on Compiler Construction* (Vienna, Austria) *(CC'06)*. 17–31. https://doi.org/10.1007/11688839_3

[65] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2016. Synergistic Analysis of Evolving Graphs. *ACM Trans. Archit. Code Optim.* 13, 4, Article 32 (Oct. 2016), 27 pages. https://doi.org/10.1145/2992784

[66] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. 237–251. https://doi.org/10.1145/3037697.3037748

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

[68] Cathrin Weiss, Cindy Rubio-González, and Ben Liblit. 2015. Database-Backed Program Analysis for Scalable Error Propagation. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. 586–597.

[69] John Whaley and Monica S. Lam. 2004. Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation* (Washington DC, USA) *(PLDI '04)*. 131–144. https://doi.org/10.1145/996841.996859

[70] Robert P. Wilson and Monica S. Lam. 1995. Efficient Context-Sensitive Pointer Analysis for C Programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation* (La Jolla, California, USA) *(PLDI '95)*. 1–12. https://doi.org/10.1145/207110.207111

[71] Meng Wu and Chao Wang. 2019. Abstract Interpretation under Speculative Execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. 802–815. https://doi.org/10.1145/3314221.3314647

[72] Yichen Xie and Alex Aiken. 2005. Scalable Error Detection Using Boolean Satisfiability. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Long Beach, California, USA) *(POPL '05)*. 351–363. https://doi.org/10.1145/1040305.1040334

[73] Guoqing Xu and Atanas Rountev. 2008. Merging Equivalent Contexts for Scalable Heap-Cloning-Based Context-Sensitive Points-to Analysis. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA '08)*. 225–236. https://doi.org/10.1145/1390630.1390658

[74] Guoqing Xu, Atanas Rountev, and Manu Sridharan. 2009. Scaling CFL-Reachability-Based Points-To Analysis Using Context-Sensitive Must-Not-Alias Analysis. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming* (Italy) *(Genoa)*. 98–122. https://doi.org/10.1007/978-3-642-03013-0_6

[75] Dacong Yan, Guoqing Xu, and Atanas Rountev. 2011. Demand-Driven Context-Sensitive Alias Analysis for Java. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (Toronto, Ontario, Canada) *(ISSTA '11)*. 155–165. https://doi.org/10.1145/2001420.2001440

[76] Greta Yorsh, Eran Yahav, and Satish Chandra. 2008. Generating Precise and Concise Procedure Summaries. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. 221–234. https://doi.org/10.1145/1328438.1328467

[77] Hongtao Yu, Jingling Xue, Wei Huo, Xiaobing Feng, and Zhaoqing Zhang. 2010. Level by Level: Making Flow- and Context-Sensitive Pointer Analysis Scalable for Millions of Lines of Code. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Toronto, Ontario, Canada) *(CGO '10)*. 218–229. https://doi.org/10.1145/1772954.1772985

[78] Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. 2014. On Abstraction Refinement for Program Analyses in Datalog. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. 239–248. https://doi.org/10.1145/2594291.2594327

[79] Jisheng Zhao, Michael G. Burke, and Vivek Sarkar. 2018. Parallel Sparse Flow-Sensitive Points-to Analysis. In *Proceedings of the 27th International Conference on Compiler Construction* (Vienna, Austria) *(CC 2018)*. 59–70. https://doi.org/10.1145/3178372.3179517

[80] Xin Zheng and Radu Rugina. 2008. Demand-Driven Alias Analysis for C. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. 197–208. https://doi.org/10.1145/1328438.1328464

[81] Z. Zuo, R. Gu, X. Jiang, Z. Wang, Y. Huang, L. Wang, and X. Li. 2019. BigSpa: An Efficient Interprocedural Static Analysis Engine in the Cloud. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 771–780. https://doi.org/10.1109/IPDPS.2019.00086

[82] Zhiqiang Zuo, John Thorpe, Yifei Wang, Qiuhong Pan, Shenming Lu, Kai Wang, Guoqing Harry Xu, Linzhang Wang, and Xuandong Li. 2019. Grapple: A Graph System for Static Finite-State Property Checking of Large-Scale Systems Code. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Article 38, 17 pages. https://doi.org/10.1145/3302424.3303972