

Improving the Efficiency of Graph-Based Static Analysis

by Yuxiang Lei

Thesis submitted in fulfilment of the requirements for
the degree of

Doctor of Philosophy (Software Engineering)

under the supervision of Dr Yulei Sui

University of Technology Sydney
Faculty of Engineering and Information Technology

12/2022

Improving the Efficiency of Graph-Based Static Analysis

*A thesis submitted in fulfilment of the requirements
for the degree of*

Doctor of Philosophy
in
Software Engineering

by

Yuxiang Lei

to

School of Computer Science
Faculty of Engineering and Information Technology
University of Technology Sydney
NSW - 2007, Australia

December 2022

© 2022 by Yuxiang Lei
All Rights Reserved

CERTIFICATE OF ORIGINAL AUTHORSHIP

I, *Yuxiang Lei* declare that this thesis is submitted in fulfilment of the requirements for the award of *Doctor of Philosophy*, in the *School of Computer Science, Faculty of Engineering and Information Technology* at the University of Technology Sydney. This thesis is wholly my own work unless otherwise referenced or acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis. This document has not been submitted for qualifications at any other academic institution. This research is supported by the Australian Government Research Training Program.

Production Note:
Signature removed prior to publication.

SIGNATURE: _____

DATE: 30th December, 2022

PLACE: Sydney, Australia

ABSTRACT

Generally speaking, static program analysis is to figure out whether a program can do whatever the program designers want it to do without actually executing the program. From different perspectives, static analysis studies various properties of a program, including correctness, robustness, liveness, safety and efficiency. As contemporary programs usually tend to be large and complex, developing efficient automatic program analysis techniques while maintaining soundness and precision is desirable.

Transitivity extensively manifests in the executions of programs, where controls and data are propagated and processed via flows. Taking data flow as an example, two assignment instructions $a = b$ and $b = c$ imply a result $a = c$, which means that the value of c flows into a via b . Static analyses inevitably include the analysis of flows, which is usually conducted in the form of solving dynamic transitive closure on the abstract graph of programs. The inefficiency arises from not only the high complexity of transitive closure itself but also the high redundancies of the analysis techniques.

This dissertation studies improving the efficiency of dynamic transitive closures on graph-based static analysis. Specifically, it focuses on improving the efficiencies of three popular static analysis frameworks: context-free language reachability, recursive state machine reachability and set constraint analysis. All the three frameworks are under the scope of graph analytics. Namely, all the analyses operate on an abstract graph of the target program.

In this dissertation, the methodologies focus more on eliminating redun-

dancy rather than theoretically lowering complexity.

For transitive redundancy that arises from the massive re-computations and re-derivations during the analysis procedures, we design a hybrid data structure and apply it to context-free language reachability. Based on this, we propose a partially ordered algorithm, which significantly improves the scalability of context-free language reachability analysis by eliminating most re-computations and re-derivations.

For trivial nodes and edges in the abstract graphs of programs, which cause extra computations in the analysis procedure, we develop a graph folding technique to remove redundant nodes and edges in the preprocessing stage and apply it to recursive state machine reachability. The graph folding technique extends the applicability of some existing techniques from particular scenarios to general analysis as long as the recursive state machine is given and is well compatible with other preprocessing techniques.

For set constraint analyses where the graph contains weighted edges, we discover the derivation equivalence property and propose an approach that avoids the infinite iterations caused by weighted cycles during constraint solving. The derivation equivalence based constraint solving is highly efficient while maintaining the precision.

Notably, the three dynamic transitive closure based program analysis frameworks, i.e., context-free language reachability, recursive state machine reachability and set constraint solving, are generally recognized as interconvertible. Accordingly, the three techniques proposed in this dissertation are mutually compatible. The empirical study on real-world clients, including value-flow analysis, alias analysis and pointer analysis, shows that our approaches are practical and effective.

ACKNOWLEDGMENTS

First, I would like to express my sincere gratitude to my supervisor, Prof. Yulei Sui, for presenting me with a chance to study at the University of Technology Sydney as a sponsored student. He told me how to improve my communication, writing and programming skills. His wisdom, dedication, scrupulousness and patience guided me throughout my PhD candidature and continuously illuminated my future research.

Besides, I would like to thank Prof. Qirun Zhang and Prof. Shinhwei Tan. Their deep insights and sweet suggestions inspired me when I got lost in my research topics.

Moreover, I would like to thank my colleagues Dr. Pei Xu, Dr. Guanqin Zhang and Dr. Xiao Cheng, who helped me with my school issues when I was on leave of absence and not in Sydney due to COVID-19. I would also like to thank Dr. Chandranath Adak, who provided this beautiful thesis template.

My greatest thankfulness is owed to my parents. They brought me up, sharing knowledge with me and supporting me to pursue a higher education degree from material and spiritual tiers. Without them, I would never have had a chance to study and meet friendly people at UTS. When I was frustrated, their encouragements were always the most powerful motivation to me.

TABLE OF CONTENTS

List of Publications	xi
List of Figures	xiii
List of Tables	xvii
1 Introduction	1
1.1 Background	1
1.1.1 Context-Free Language Reachability (CFL-Reachability)	3
1.1.2 Recursive Static Machine (RSM)	4
1.1.3 Set Constraint Analysis	5
1.2 Research Topics	6
1.2.1 Eliminating Transitive Redundancy On-The-Fly	7
1.2.2 Simplifying the Input Graph in the Proprocessing Stage	9
1.2.3 Precise and Efficient Handling of Positive Weight Cycles	10
1.3 Contributions	13
1.3.1 Eliminating Transitive Redundancy On-The-Fly	13
1.3.2 Recursive State Machine Guided Graph Folding	13
1.3.3 Fast and Precise Handling of Positive Weight Cycles	14
1.4 Thesis Organization	15
2 Literature Review	17
2.1 Context-Free Language and CFL-reachability	17

2.2	Pushdown Automata and Recursive State Machines	18
2.3	Set Constraint Analysis	19
2.4	Efficiency Study	20
3	Taming Transitive Redundancy for Context-Free Language Reachability	23
3.1	Problem Formulation	23
3.1.1	CFL-Reachability	23
3.1.2	Redundant Derivations and Transitive Redundancy	26
3.1.3	Research Problem	28
3.2	Our Solution	30
3.2.1	Hybrid Graph Representation for Reducing Redundant Derivations	31
3.2.2	Dynamic Construction of Spanning Trees	35
3.2.3	POCR: A Fast Partially Ordered CFL-Reachability Algorithm for All-Pairs Analyses	41
3.3	Discussion: Effectiveness of POCR	43
3.3.1	Grammars Benefiting from POCR	43
3.3.2	Grammar-Driven Redundancy Reduction	44
3.4	Experimental Evaluation	46
3.4.1	Experimental Setup	47
3.4.2	RQ 1: Reduction of Redundant Derivations	49
3.4.3	RQ 2: Speedups Over Baselines	54
3.4.4	RQ 3: POCR vs. Grammar Rewriting	56
3.4.5	Summary	58
4	Recursive State Machine Guided Graph Folding	61
4.1	Problem Formulation	61
4.1.1	Recursive Static Machine	62
4.1.2	RSM-Reachability	64

4.1.3	Research Problem	65
4.2	Principle for Graph Folding	67
4.2.1	Correspondences in Graph Folding and RSM-Reachability	68
4.2.2	Folding Principle	71
4.2.3	Correctness of Folding Principle	75
4.3	Graph-Folding Algorithm	79
4.3.1	Identifying Foldable Node Pairs	79
4.3.2	Overall Algorithm	82
4.4	Experiment	83
4.4.1	Experimental Setup	84
4.4.2	Performance in Reducing Graph Sizes	85
4.4.3	Speedup and Memory Overhead	89
4.4.4	Discussions	92
4.4.5	Summary	92
5	Derivation Equivalence Based Set Constraint Solving	95
5.1	Problem Formulation	95
5.1.1	Pointer Analysis in Set Constraints	95
5.1.2	Field-Sensitivity and Positive Weight Cycles	97
5.1.3	Derivation Equivalence Based Constraint Solving	101
5.2	Our Solution	103
5.2.1	Stride-based Field Representation	103
5.2.2	Inference Rules	105
5.2.3	DEA: a Derivation Equivalence Algorithm	110
5.3	Implementation of Field-Sensitive Pointer Analysis for C/C++	112
5.4	Experimental Evaluation	114
5.4.1	Experimental Setup	114
5.4.2	Results and Analysis	115

TABLE OF CONTENTS

5.4.3 Summary	121
6 Conclusion and Future Works	123
A Appendix	125
A.1 Proof of the Soundness of Algorithm 2	125
A.2 Proof of Property 4.2	126
A.3 Proof of Property 4.3	127
Bibliography	133

LIST OF PUBLICATIONS

The following is a list of publications that are included in this thesis.

Chapter 3:

Yuxiang Lei, Yulei Sui, Shuo Ding, and Qirun Zhang. "Taming transitive redundancy for context-free language reachability." *Proceedings of the ACM on Programming Languages* 6. *OOPSLA2 (2022)*: 1556-1582. (SIPLAN 2022 Distinguished Artifact Award)

Chapter 4:

Yuxiang Lei, Yulei Sui, Qirun Zhang, and Shinhwei Tan. "Recursive State Machine Guided Graph Folding for Context-Free Language Reachability." *Proceedings of the ACM on Programming Languages* 7. *PLDI (2023)*.

Chapter 5:

Yuxiang Lei and Yulei Sui. "Fast and Precise Handling of Positive Weight Cycles for Field-sensitive Pointer Analysis." *Static Analysis: 26th International Symposium, SAS (2019)*, *Proceedings* 26 (pp. 27-47). (Radhia Cousot Young Researcher Best Paper Award)

LIST OF FIGURES

FIGURE	Page
1.1 A example of value-flow analysis, where the value-flow from p to r is unknown unless the value-flow relations of the variables in the function foo is determined.	3
1.2 Transitive redundancy caused by two ways to derive $v_1 \xrightarrow{A} v_4$ from $v_1 \xrightarrow{A} v_2 \xrightarrow{A} v_3 \xrightarrow{A} v_4$.	8
1.3 Positive weight cycles and infinite derivations in CFL-reachability.	11
1.4 Positive weight cycles and infinite derivations in set constraint analysis.	11
1.5 Main contributions of this thesis.	14
3.1 Redundant checks in partial transitive derivations, where A is a transitive relation.	27
3.2 An example.	28
3.3 Predecessor trees and successor trees. The root of each spanning tree is circled.	32
3.4 Processing a new edge $v_2 \xrightarrow{A} v_3$ added to Figure 3.3(a) by $NewTrEdge(A, v_2, v_3)$, which traverses $ptree(A, v_2)$ and $stree(A, v_3)$, and updates the $ptrees$ and $strees$ of the visited nodes. In each step, nodes being visited and newly added are marked in red and blue, respectively.	38
3.5 CFG for context-sensitive value-flow analysis.	48
3.6 CFG for field-sensitive alias analysis.	48

3.7 The computational redundancy of the three approaches in solving the two clients. The value is computed by (#Deriv / #Add). The vertical axis is logarithmic. The peak, valley and average values of each approach are marked in the charts. 53

3.8 Extra memory overhead of POCR over the standard algorithm. Only the benchmarks successfully solved by the standard algorithm are considered. . . 56

3.9 CFG modified from Figure 3.6 by removing the doubly recursive rules. 57

3.10 Results of context-sensitive value-flow analysis using the modified grammar in Figure 3.9(a). 58

3.11 Results of field-sensitive alias analysis using the modified grammar in Figure 3.9(b). 59

4.1 An example of RSM-reachability and graph folding. 62

4.2 Correspondences between paths before and after folding. Without loss of generality, we assume that y is merged into x in the xy -folded graph. 68

4.3 A path p_G and its corresponding transition chain p_R before and after folding (x, y) . Without loss of generality, we assume that y is merged into x in the xy -folded graph. 70

4.4 Example of subsumption and equivalence relations of states and instances of foldable pairs (x, y) 73

4.5 Rules for *Cond. 2*, where $\alpha \in \{0, 1, 2\}$, L_{xy} , $L_{x_}$, $L_{y\cancel{x}}$, and $Nr(L_{x_})$ are defined in Table 4.2 and Eq. 4.1. 73

4.6 For the problem running on the RSM in Figure 4.1(a), if $v_1 \in V_{src}$ and $v_2 \in V_{snk}$, folding (x, y) introduces an additional reachable pair (v_1, v_2) in G' via $v_1 \xrightarrow{q} x \xrightarrow{d} v_2$, which violates reachability equivalence. 76

4.7 Four basic types of xy -FEQ classes, where each type contains at most one xy -subpath. For simplicity, we only draw one edge from x to y and one edge from y to x 77

4.8	The CFG and RSM for C/C++ context-sensitive value-flow analysis.	84
4.9	The RSM for C/C++ field-sensitive alias analysis, where the nodes in double circle denote the exits of the box.	84
4.10	Reduction rates of nodes and edges in the input graphs of value-flow analysis.	87
4.11	Reduction rates of nodes and edges in the input graphs of alias analysis, where GF denotes graph folding, SCC denotes cycle elimination and InterDyck denotes the InterDyck graph simplification.	88
4.12	Speedups of CFL-reachability by GF, SCC, InterDyck and their combinations.	90
4.13	Reduction rates of memory overhead by GF.	91
4.14	Two instances of foldable node pair (x, y) in the problem running upon the RSM of Figure 4.4.	91
5.1	Flowchart of dynamic transitive closure algorithm for pointer analysis.	97
5.2	Rules for field-sensitive pointer analysis.	98
5.3	Positive weight cycle and infinite derivations.	100
5.4	An example.	102
5.5	SFR-based rules for field-sensitive pointer analysis.	106
5.6	Solving $p_1 \xrightarrow{\text{Field}_2} p_2$, which resides in multiple cycles, with SFRs.	107
5.7	Solving $q \xrightarrow{\text{Store}} p$ and $p \xrightarrow{\text{Load}} r$ for overlapping SFRs.	109
5.8	C code fragment and its LLVM IR.	113
5.9	Percentages of fields derived when solving PWCs out of the total number of fields, i.e., $\frac{\#FieldByPWC}{\#Field} * 100$	117
5.10	Comparing the time distribution of the three analysis phases of DEA with that of baseline (normalized with baseline as the base).	120
A.15	The above four rules holds if the rules in Figure 5 hold for all $\alpha \in \{0, 1, 2\}$	128

LIST OF TABLES

TABLE	Page
3.1 Benchmark info. #Node and #Edge respectively denote the number of nodes and edges in the initial graphs.	49
3.2 Result of context-sensitive value-flow analysis. #Add(k) and #Deriv(k) denotes the number of edges added to the graph and created when solving CFL-reachability, measured in thousands. Reduction(%) denotes the reduction rate of redundant derivations of GSA and POCR. Time(s) denotes the runtime of each approach, measured in seconds. The baselines of both Reduction(%) and speedup are the columns “Base”.	51
3.3 Result of field-sensitive alias analysis. #Add(k) and #Deriv(k) denotes the number of edges added to the graph and created when solving CFL-reachability, measured in thousands. Reduction(%) denotes the reduction rate of redundant derivations of GSA and POCR. Time(s) denotes the runtime of each approach, measured in seconds. The baselines of both Reduction(%) and speedup are the columns “Base”.	52
4.1 The solutions of the RSM-reachability problem in G and G' of Figure 4.1(b). .	66
4.2 Edge label notations for discussing RSM-reachability.	71

4.3	Benchmark info and results of the baseline. #Node and #Edge denote the number of nodes and edges of each input graph. P-Edge% denotes the percentage of parenthesis edges out of total edges of each input graph. Time/s and Mem./GB denote the runtime and memory overhead of the baseline for analyzing each program, measured in seconds and gigabytes, respectively. . .	86
4.4	Runtime of GF, SCC and InterDyck, measured in seconds.	89
5.1	Program instructions, constraints and edges.	96
5.2	Basic characteristics of the benchmarks (IR's lines of code, number of pointers, number of five types of instructions on the initial constraint graph, and maximum number of fields of the largest struct in each program).	115
5.3	Comparing the results produced by DEA with those by baseline, including the total number of address-taken variables, number of fields and the number of fields derived when resolving PWCs, and the number of Copy edges connected to/from the field object nodes derived when resolving PWCs	116
5.4	Constraint graph information (#NodeInPWC denotes the number of nodes involving PWCs by baseline; #SFR denotes the number of stride-based field representatives, generated by DEA; #CopyByPWC, denotes the number of Copy edges flowing into and going out of fields derived when solving PWCs; #CopyProcessed denotes the number of processing Copy edges.)	118
5.5	Total analysis times and the times of the three analysis stages, including CycleDec cycle detection (Lines 5–6 of Algorithm 9), PtsProp, propagating point-to information via Copy and Field edges (Lines 9–17), ProcessLdSt, adding new Copy edges when processing Load/Store and update callgraph (Lines 20–27).	119